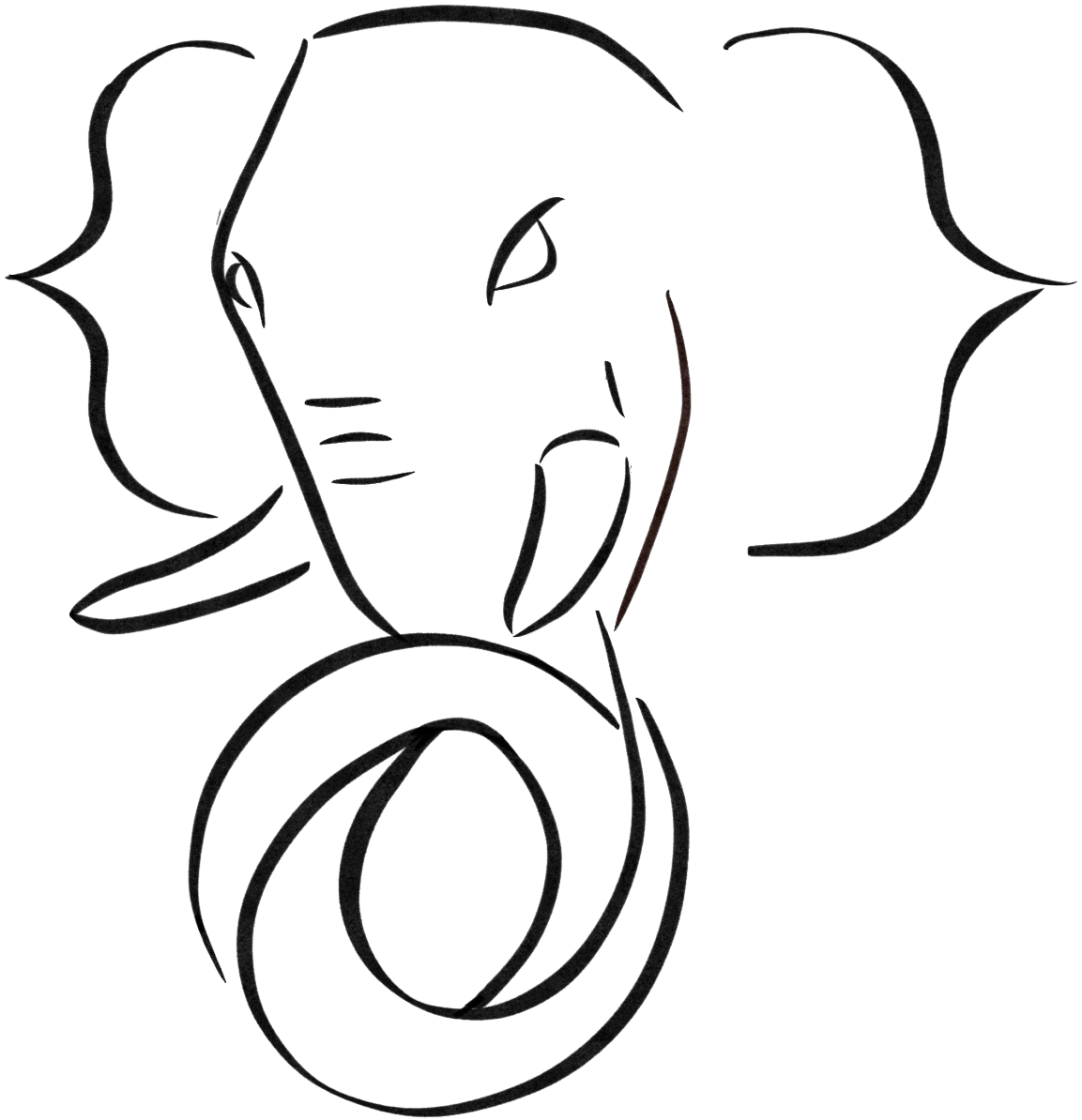
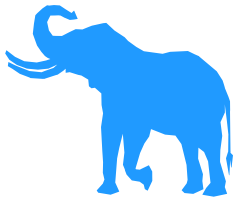




PGConf.Online 2021

# JSONB изнутри

Oleg Bartunov
Nikita Glukhov

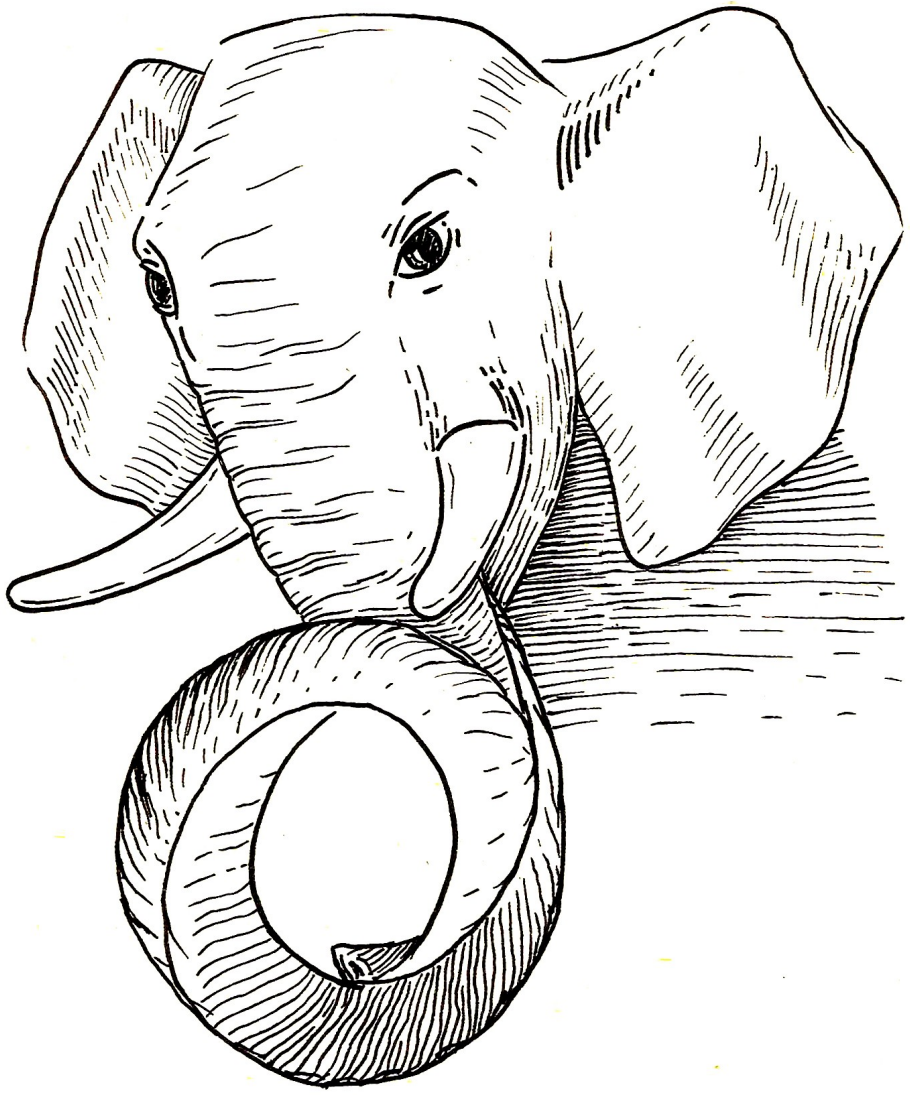
Postgres PROFESSIONAL

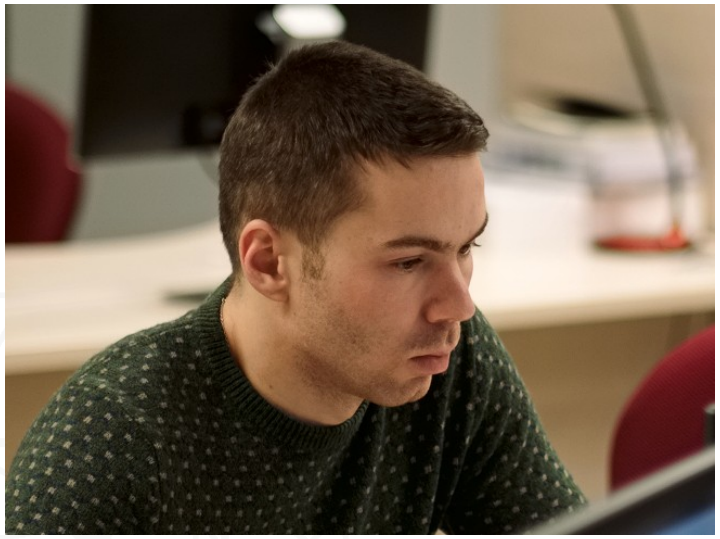
# Since Postgres95


Research scientist @
Moscow University
CEO Postgres Professional
Major PostgreSQL contributor

PostgresPro

# Nikita Glukhov





Senior developer @Postgres Professional
PostgreSQL contributor

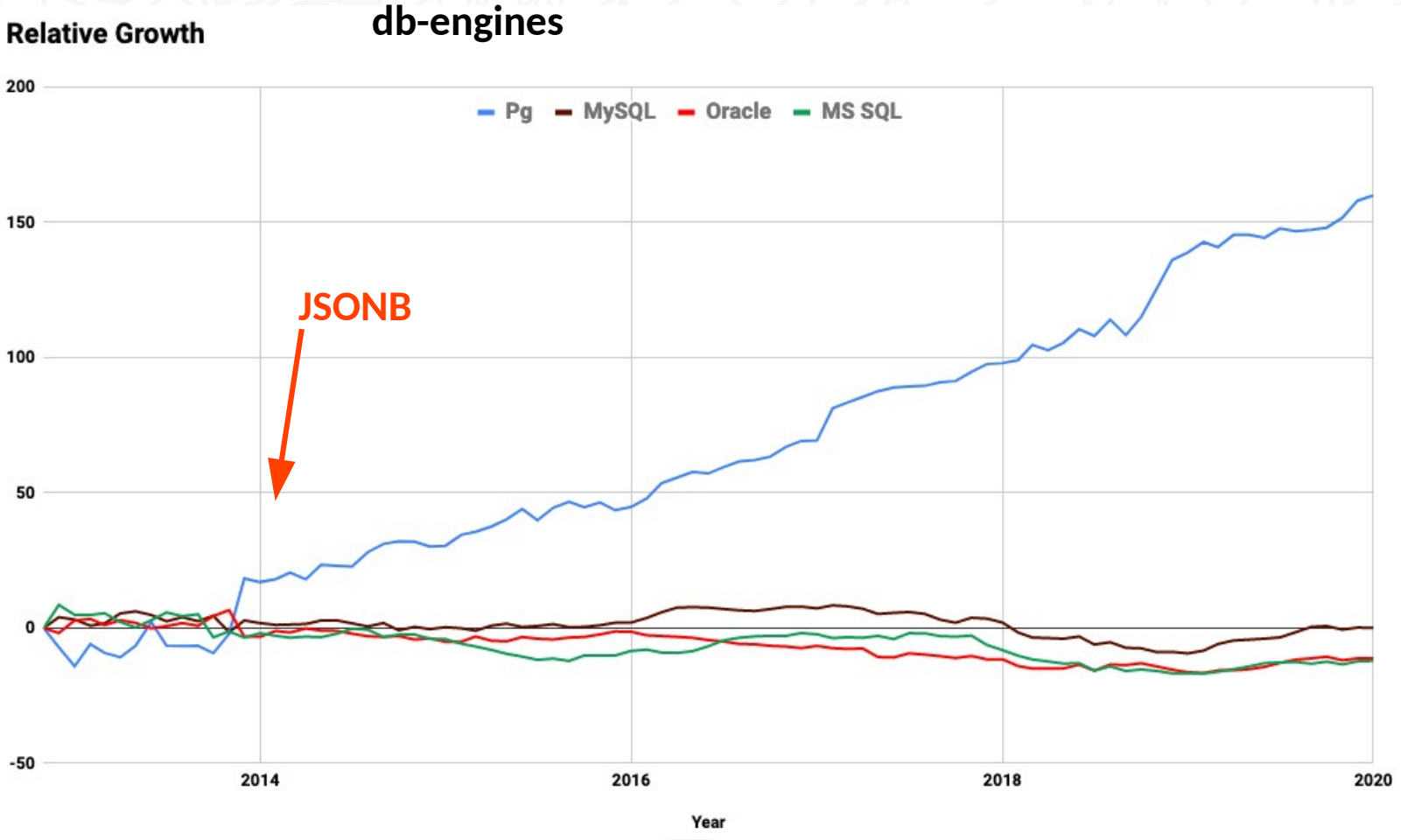
**Major CORE contributions:**
Jsonb improvements
SQL/JSON (Jsonpath)
KNN SP-GiST
Opclass parameters

**Current development:**
SQL/JSON functions
Jsonb performance

PostgresPro

# Postgres revolution: embracing relational databases

- NoSQL users attracted by the NoSQL Postgres features





**Dec 18, 2014**

PostgresPro

The goal:

JSONB - 1st-class citizen in Postgres

- Efficient storage,select, update, API

PostgresPro

# Reality: Unpredictable performance of jsonb

Small update cause 10 times slowdown !

```
CREATE TABLE test (jb jsonb);
ALTER TABLE test ALTER COLUMN jb SET STORAGE EXTERNAL;
INSERT INTO test
SELECT
  jsonb_build_object(
  'id', i,
  'foo', (select jsonb_agg(0) from generate_series(1, 1960/12)) -- [0,0,0, ...]
  ) jb
FROM
  generate_series(1, 10000) i;
```

```
=# EXPLAIN(ANALYZE, BUFFERS) SELECT jb->'id' FROM test;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..2625.00 rows=10000 width=32) (actual time=0.014..6.128 rows=10000 loops=1)
   Buffers: shared hit=2500
 Planning:
   Buffers: shared hit=5
 Planning Time: 0.087 ms
 Execution Time: 6.583 ms
(6 rows)

 =# EXPLAIN (ANALYZE, BUFFERS) UPDATE test SET jb = jb || '{"bar": "baz"}';

=# EXPLAIN (ANALYZE, BUFFERS) SELECT jb->'id' FROM test;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..2675.40 rows=10192 width=32) (actual time=0.067..65.511 rows=10000 loops=1)
   Buffers: shared hit=32548
 Planning Time: 0.044 ms
 Execution Time: 66.889 ms
(4 rows)
```

PostgresPro

# Motivational example (synthetic test)

- A table with 100 jsonbs of different sizes (130B-13MB, compressed to 130B-247KB):

```
CREATE TABLE test_toast AS
SELECT
 i id,
 jsonb_build_object(
  'key1', i,
  'key2', (select jsonb_agg(0) from
           generate_series(1, pow(10, 1 + 5.0 * i / 100.0)::int)),-- 10-100k elems
  'key3', i,
  'key4', (select jsonb_agg(0) from
           generate_series(1, pow(10, 0 + 5.0 * i / 100.0)::int)) -- 1-10k elems

 ) jb
FROM generate_series(1, 100) i;
```

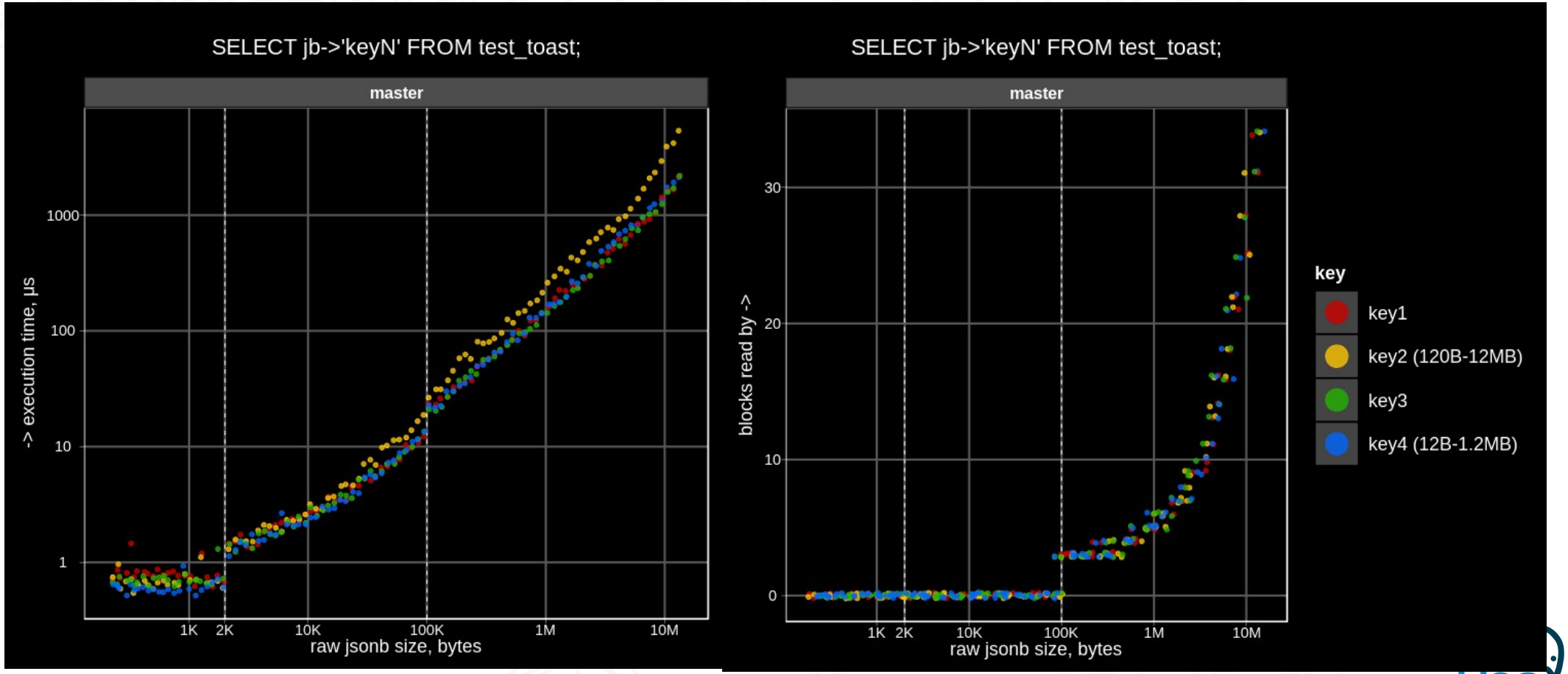
- Each jsonb looks like: `key1`, looooong `key2`, `key3`, long `key4`.
- We measure execution time of operator `->(jsonb, text)` for each row by repeating it 1000 times in the query:

```
SELECT jb -> 'keyN', jb -> 'keyN', … jb -> 'keyN' FROM test_toast WHERE id = ?;
```

PostgresPro

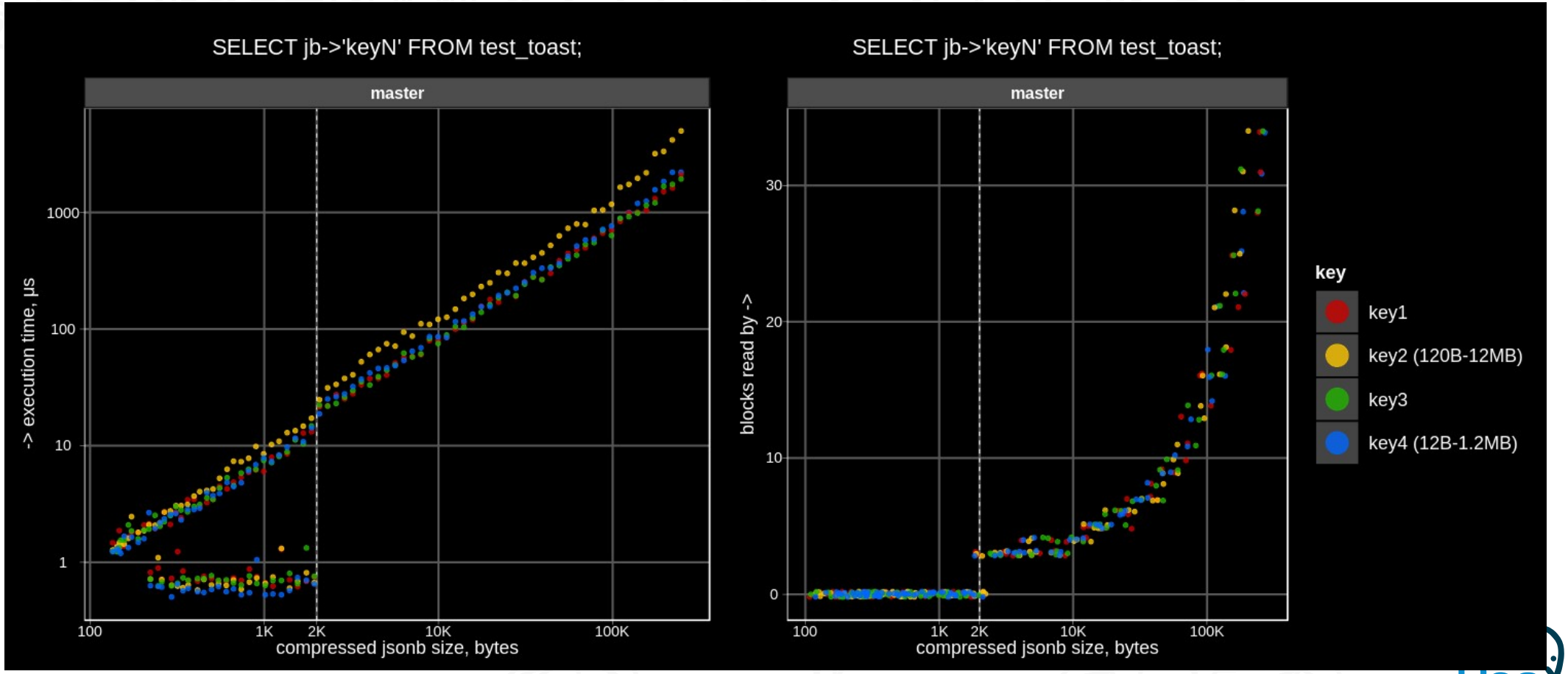
# Motivational example (synthetic test)

Key access time for TOASTed jsonbs  linearly increase with jsonb size, regardless of key size and position.



# TOAST performance problems (synthetic test)

Key access time for TOASTed jsonbs  linearly increase with jsonb size, regardless of key size and position.



# Motivational example (IMDB test)

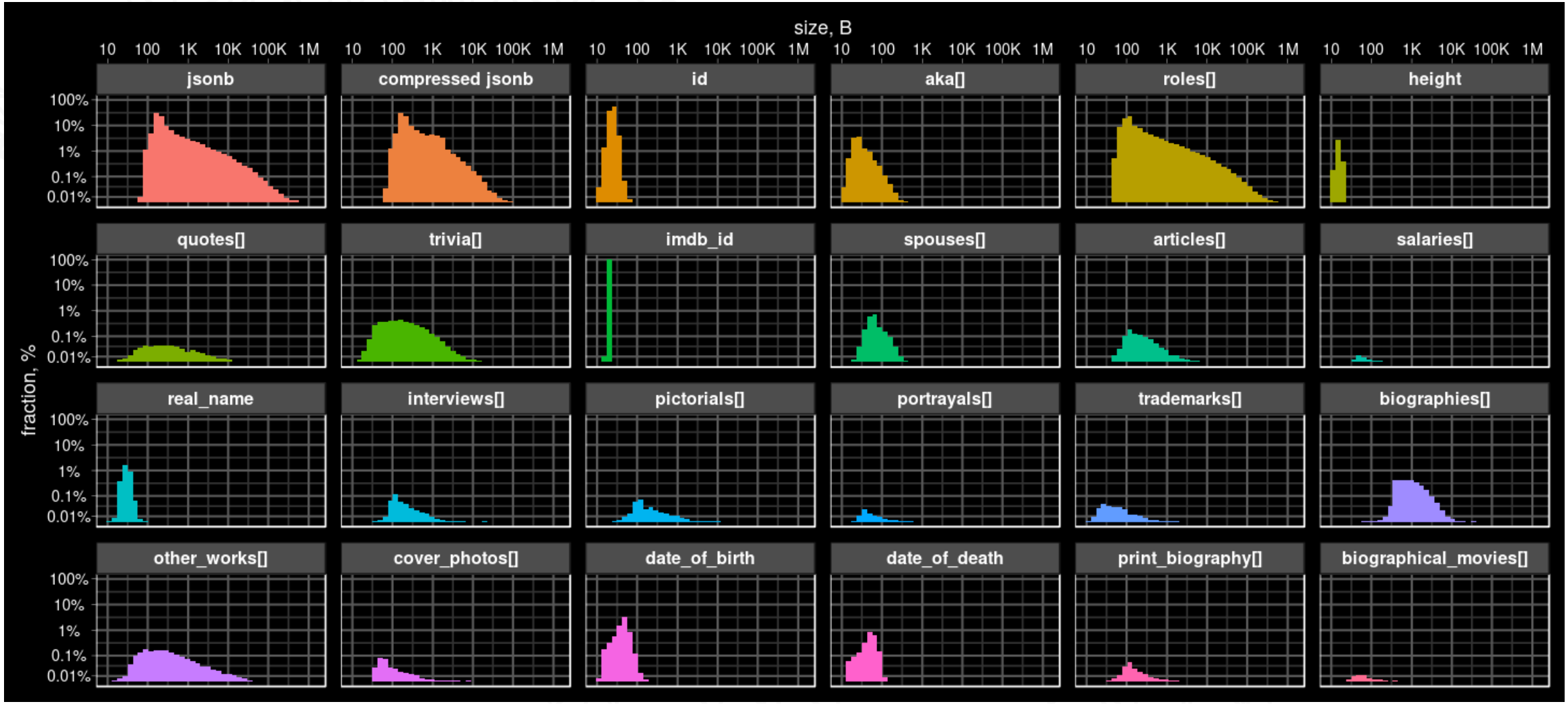
- Real-world JSON data extracted from IMDB database (imdb-22-04-2018-json.dump.gz)
- Typical IMDB «name» document looks like:

```
{
    "id": "Connors, Steve (V)",
    "roles": [
        {
            "role": "actor",
            "title": "Copperhead Creek (????)"
        },
        {
            "role": "actor",
            "title": "Ride the Wanted Trail (????)"+
        }
     ],
     "imdb_id": 1234567
 }
```

- There are many other infrequent fields, but only `id`, `imdb_id` are mandatory, and `roles` array is the **biggest** and most frequent (see next slide).

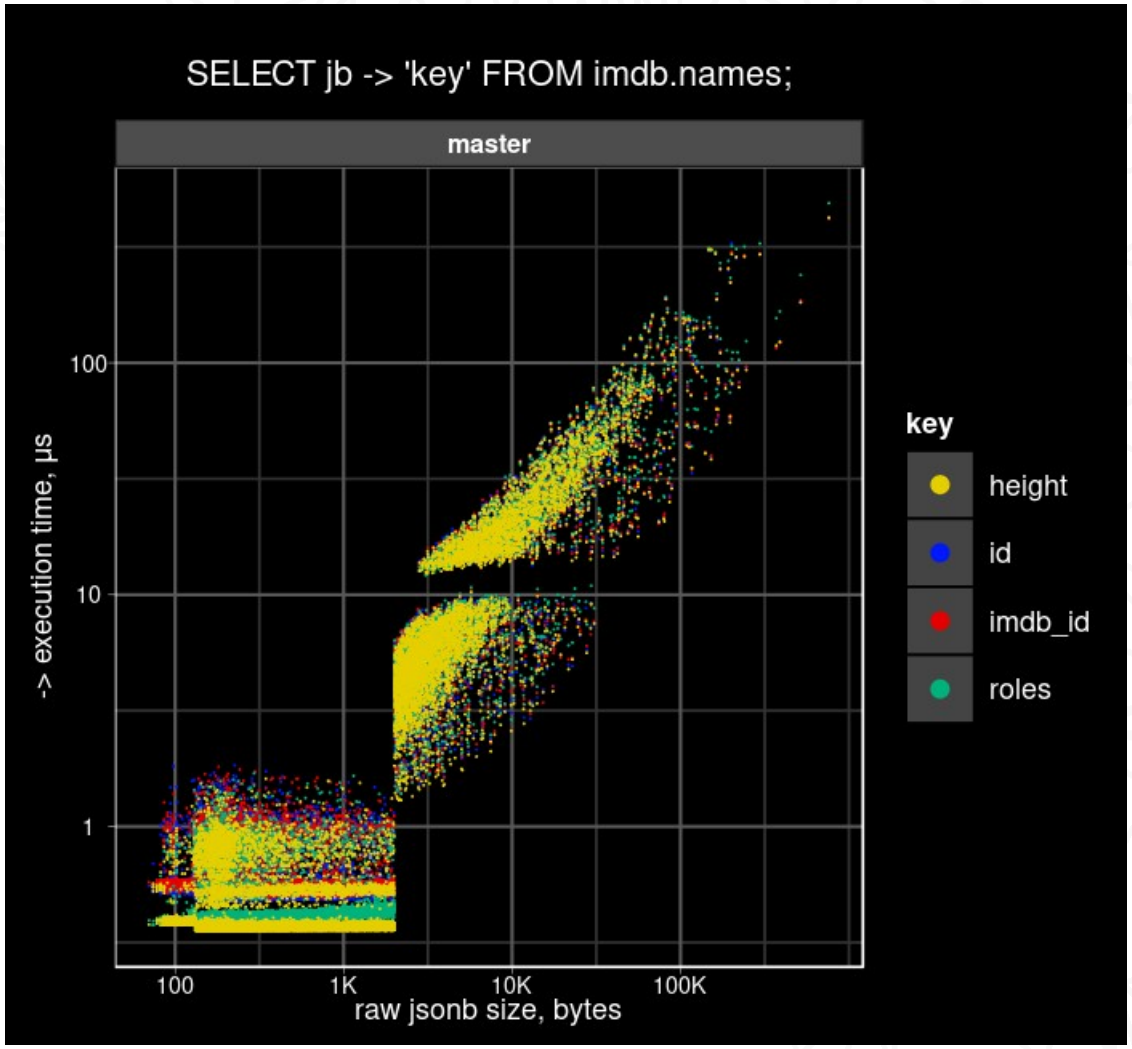
PostgresPro

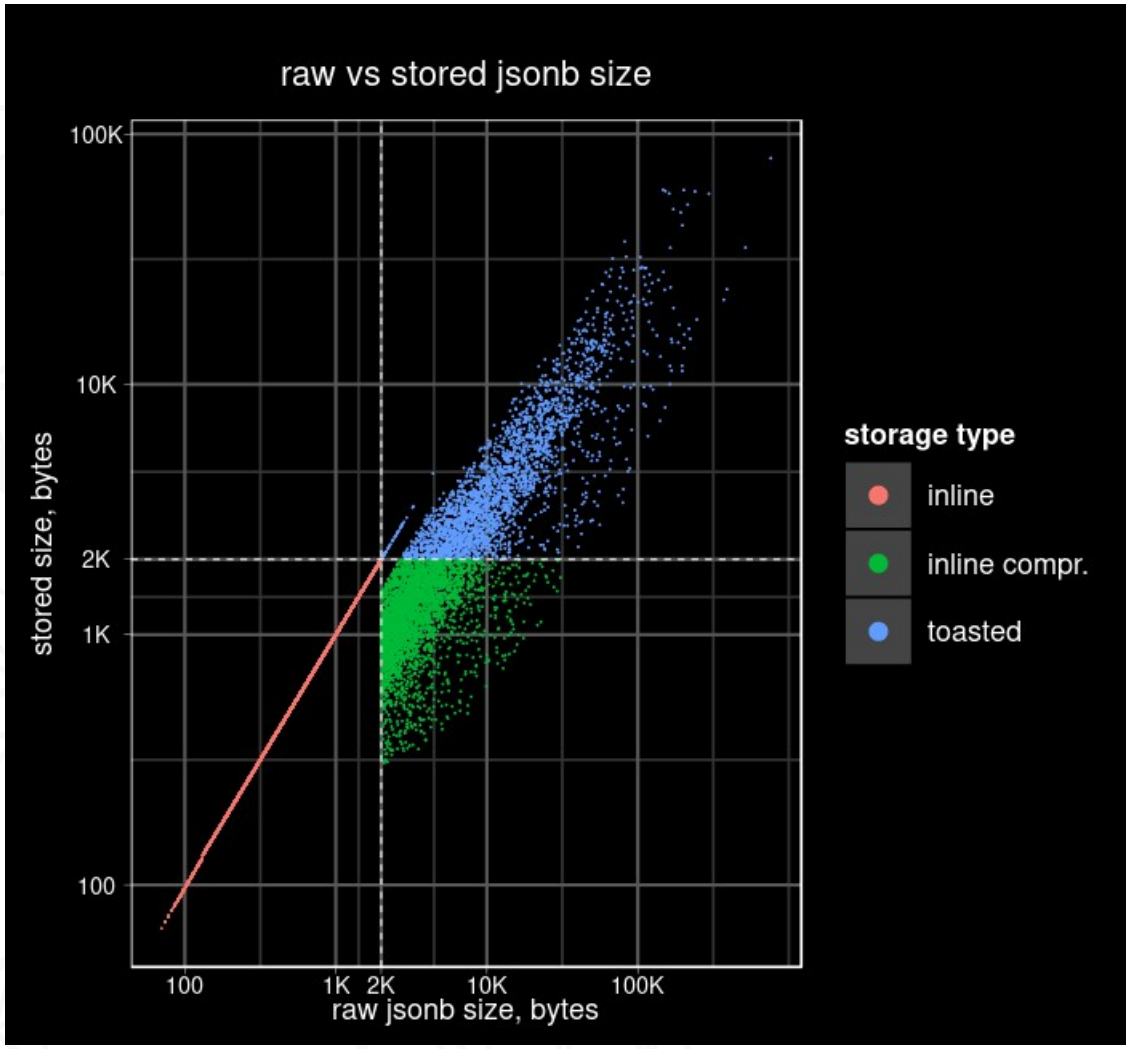
# IMDB data set field statistics




PostgresPro

# Motivational example (IMDB test)




PostgresPro

# Motivation

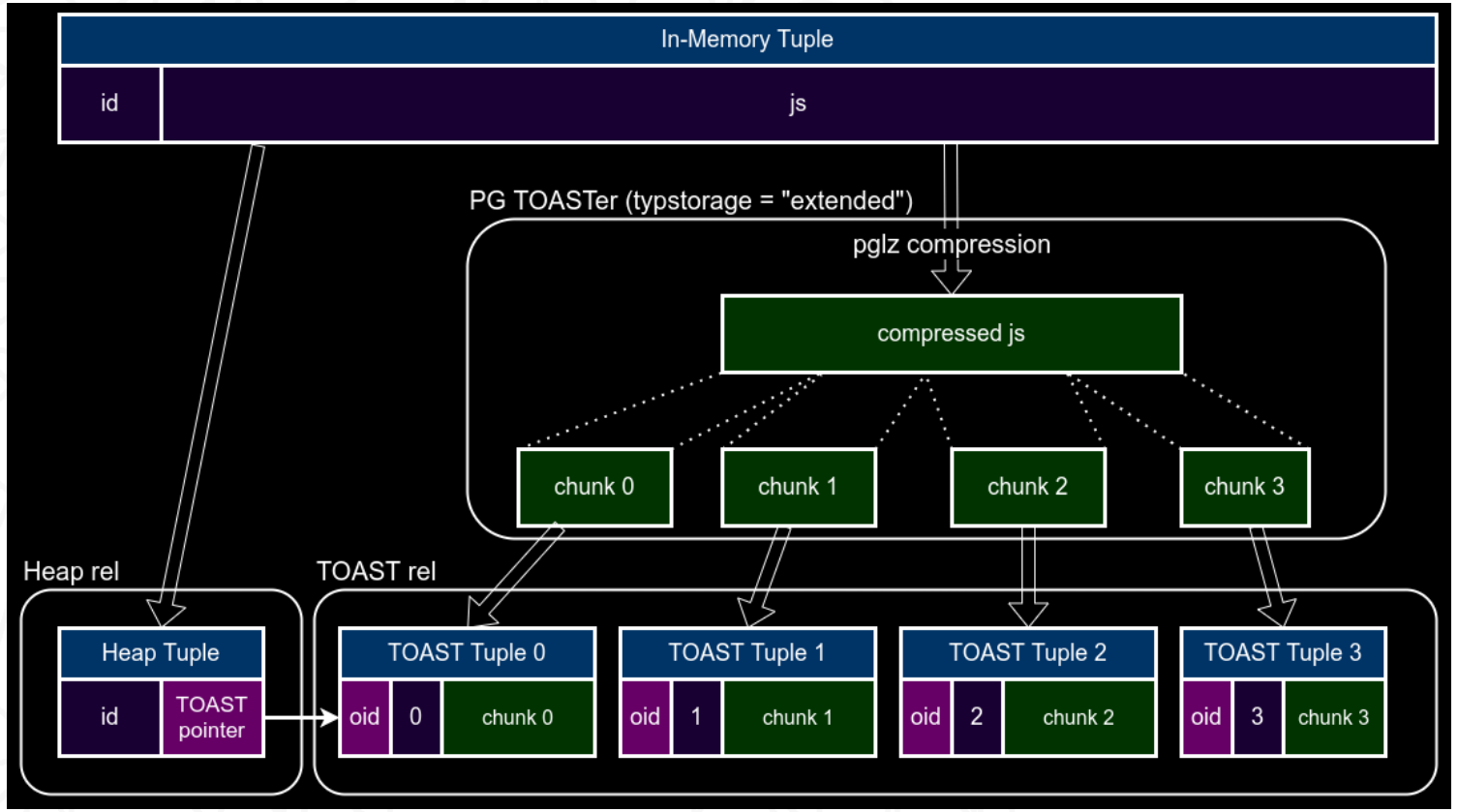
- Decompression is the biggest problem. Big overhead of decompression of the whole jsonb limits the applicability of jsonb as  document storage with partial access.
  - Need partial decompression
- Toast introduces additional overhead - read too many block
  - Read only needed blocks — partial detoast

PostgresPro

# TOAST Explained

The Oversized-Attribute Storage Technique

- TOASTed value is pglz compressed
- Compressed value is splitted into the fixed-size TOAST chunks (1996B for 8KB page)
- TOAST chunks (along with generated Oid `chunk_id` and sequnce number `chunk_seq`) stored in special TOAST relation `pg_toast.pg_toast_XXX`, created for each table containing TOASTable attributes
- Attribute in the original heap tuple is replaced with TOAST pointer (18 bytes) containing `chunk_id, toast_relid, raw_size, compressed_size`

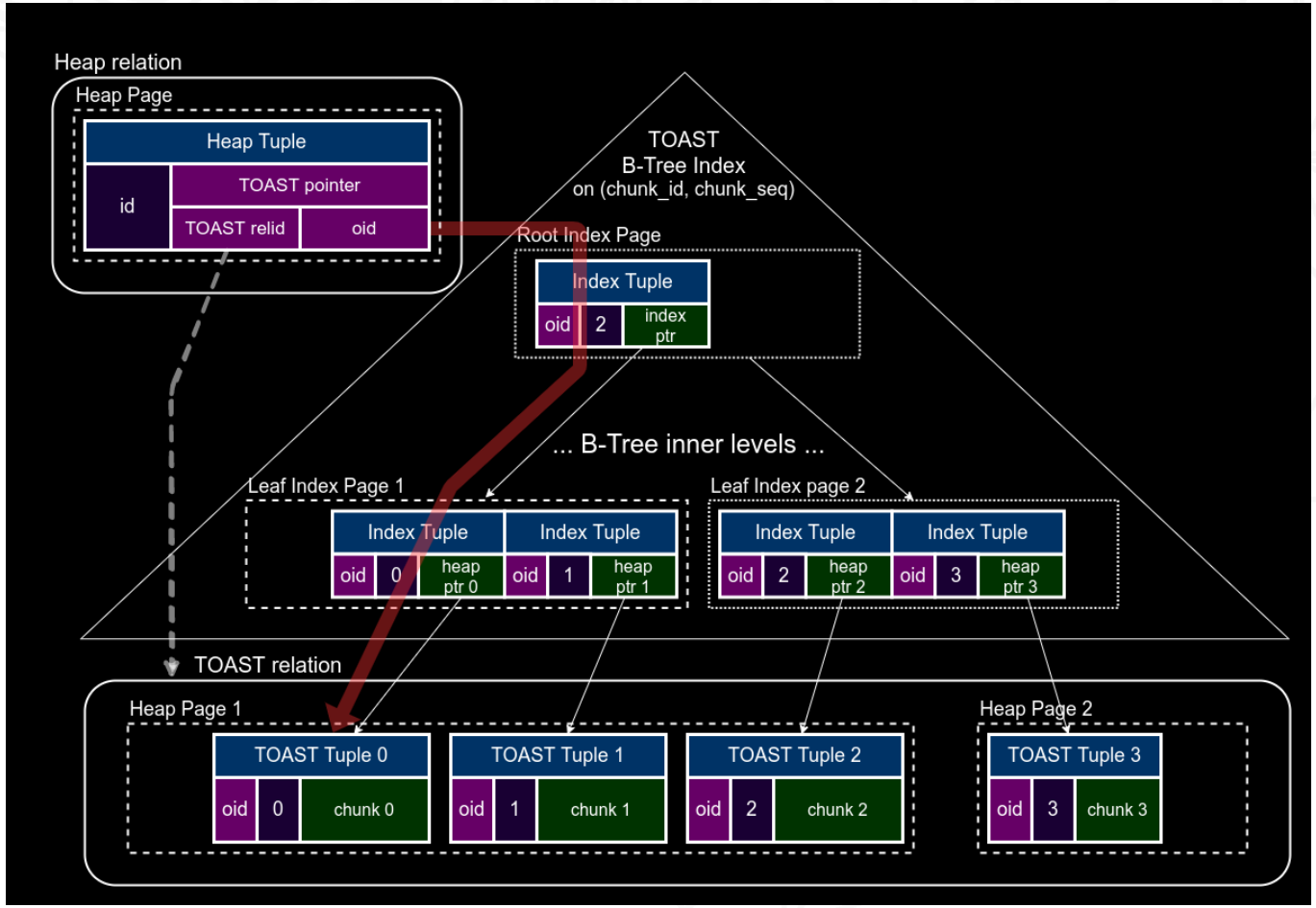



https://www.postgresql.org/docs/current/storage-toast.html

PostgresPro

# TOAST access

- TOAST pointers does not refer to heap tuples with chunks directly. Instead they contains Oid `chunk_id` and we need to descent by index (`chunk_id`, `chunk_seq`).




Overhead to read only a few bytes from the first chunk is 3,4 or even 5 additional index blocks.

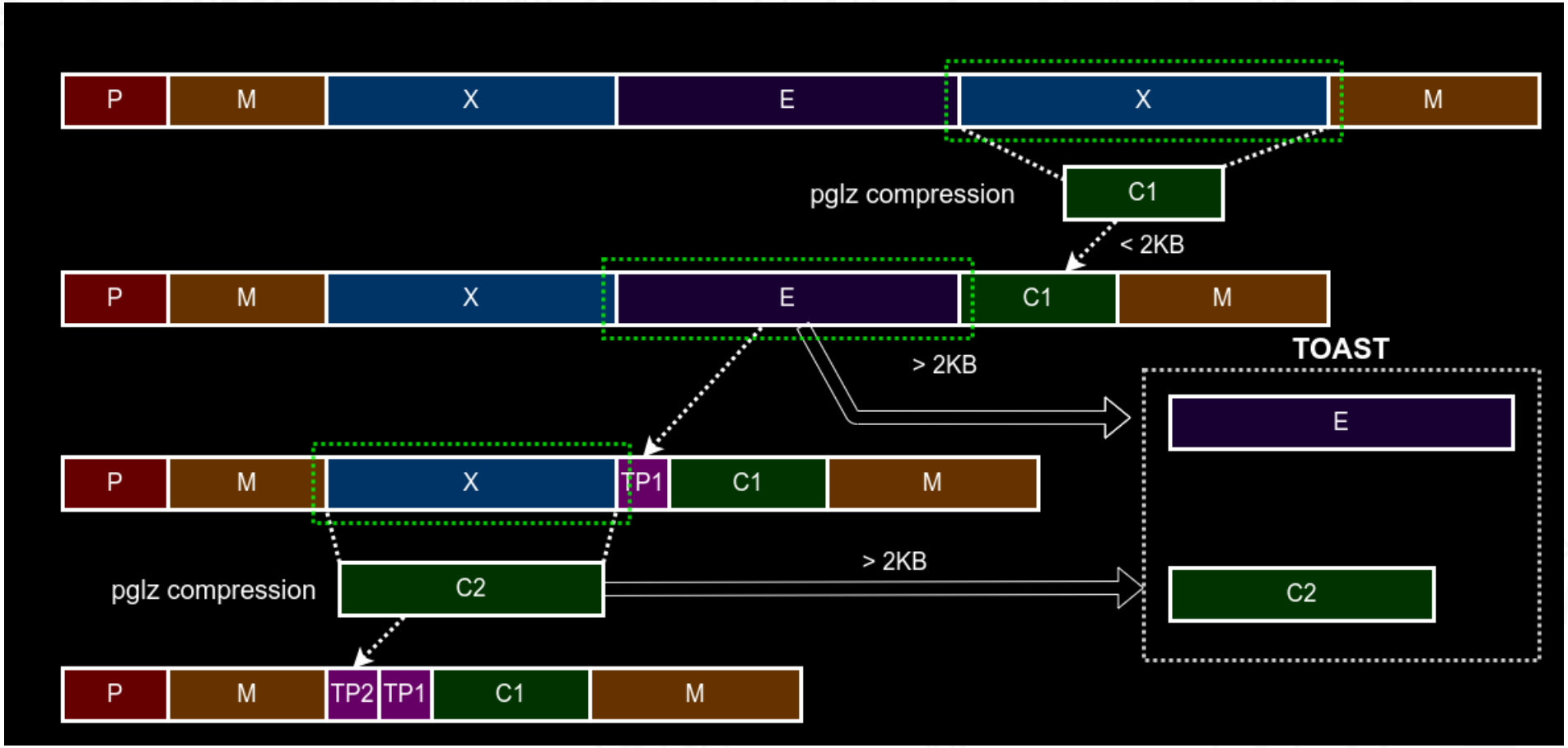
PostgresPro

# TOAST passes

- Tuple is TOASTed if its size is more than 2KB (1/4 of page size).
- There are 4 TOAST passes.
- At the each pass considered only attributes of the specific storage type (extended/external or main) starting from the largest one.
- Plain attributes are not TOASTed and not compressed at all.
- The process can stop at every step, if the resulting tuple size becomes less than 2KB.
- If the attributes were copied from the other table, they can already be compressed or TOASTed.
- TOASTed attributes are replaced with TOAST pointers.

# TOAST pass #1

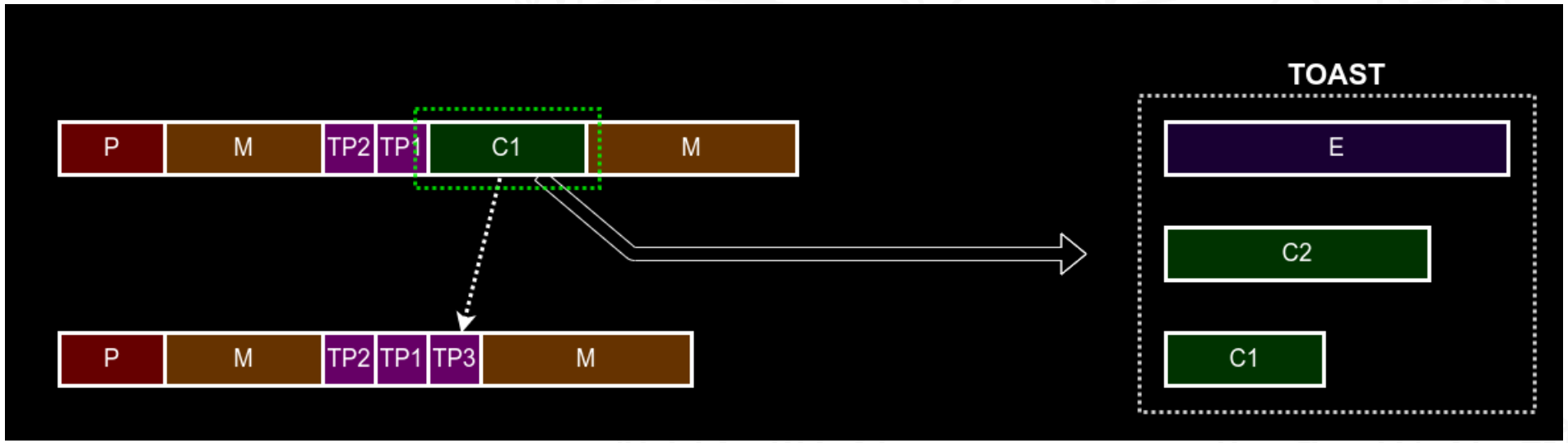
- Only "extended" and "external" attributes are considered, "extended" attributes are compressed. If their size is more than 2KB, they are TOASTed.




PostgresPro

# TOAST pass #2

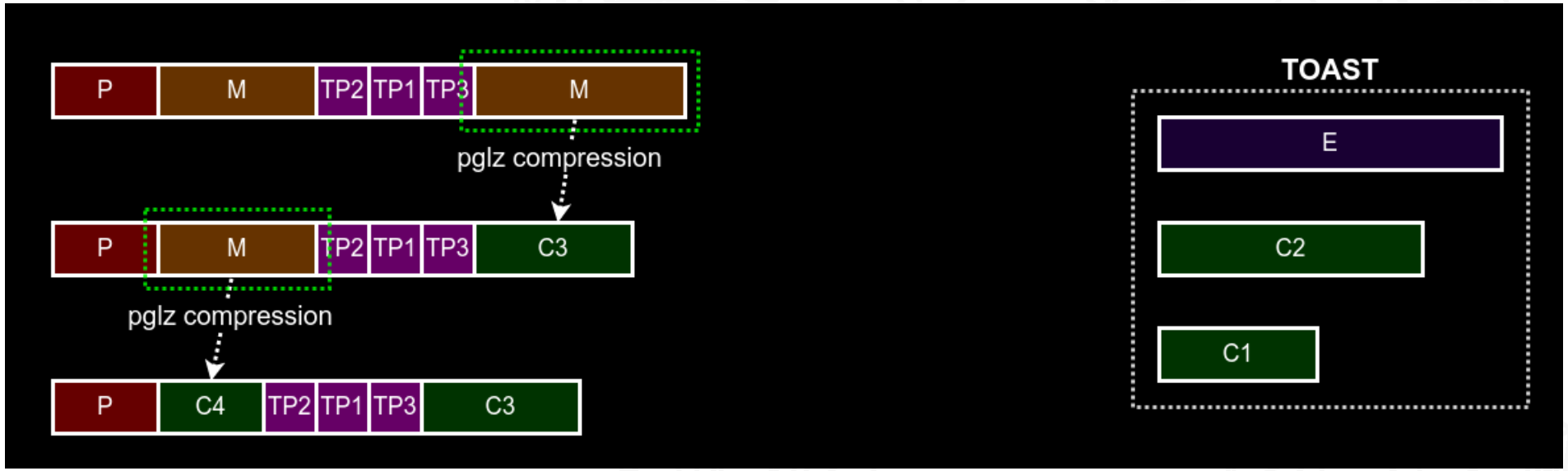
- Only "extended" and "external" attributes (that were not TOASTed in the previous pass) are considered.
- Each attribute is TOASTed, until the resulting tuple size < 2KB.




PostgresPro

# TOAST pass #3

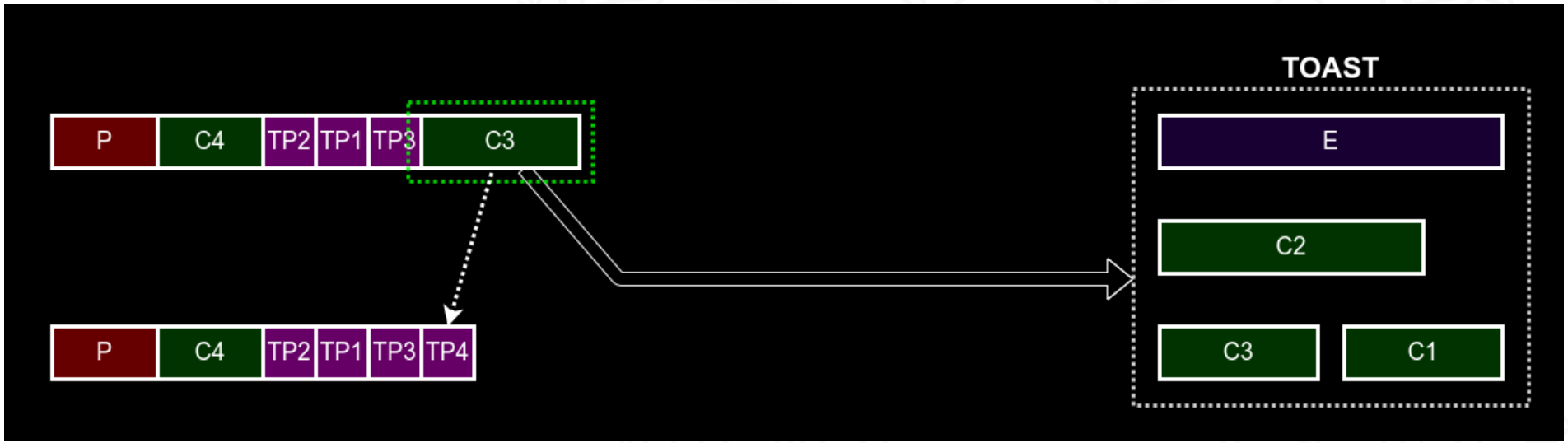
- Only "main" attributes are considered.
- Each attribute is compresed, until the resulting tuple size < 2KB.




PostgresPro

# TOAST pass #4

- Only "main" attributes are considered.
- Each attribute is TOASTed, until the resulting tuple size < 2KB.




PostgresPro

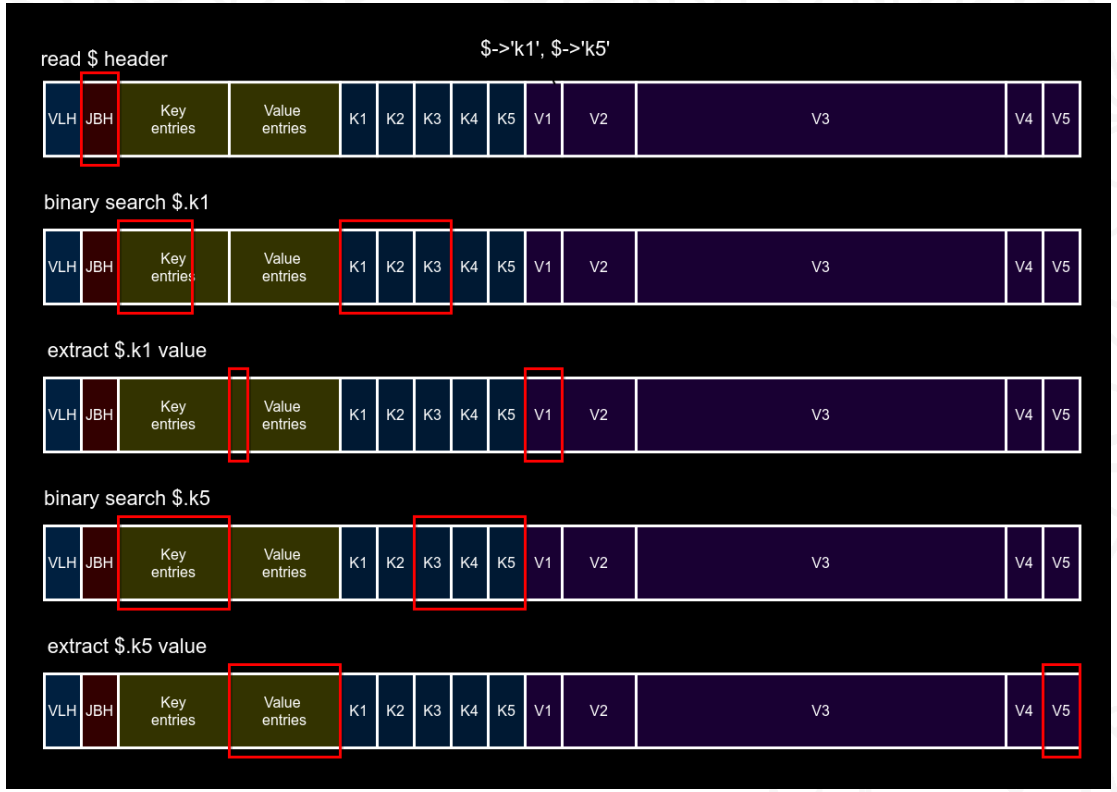
# Jsonb deTOAST improvements

- Partial pglz decompression
- Sort jsonb object key by their length
- Partial deTOASTing using TOAST iterators
- Inline TOAST
- Shared TOAST

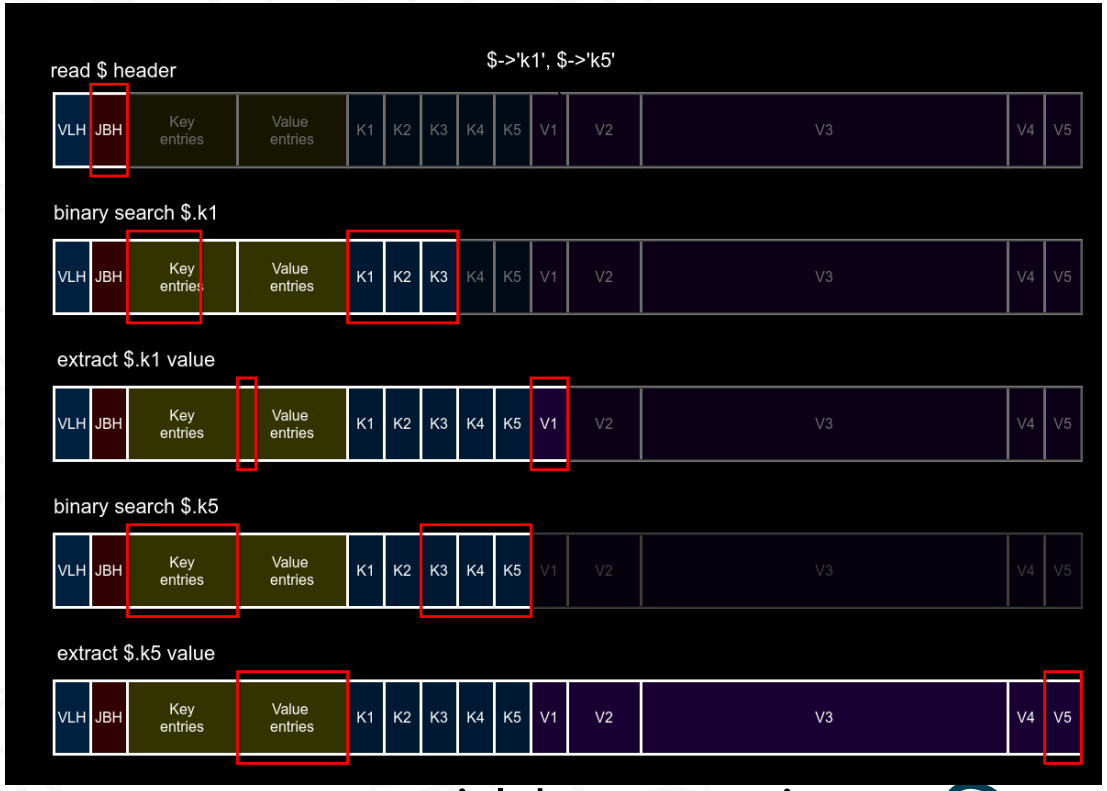
PostgresPro

# Jsonb partial decompression

- Partial decompression eliminates  overhead of pglz decompression of the whole jsonb.
- Jsonb is decompressed step by step: header, KV entries array, key name and key value. Only prefix of jsonb has to be decompressed to acccess a given key !




full decompression



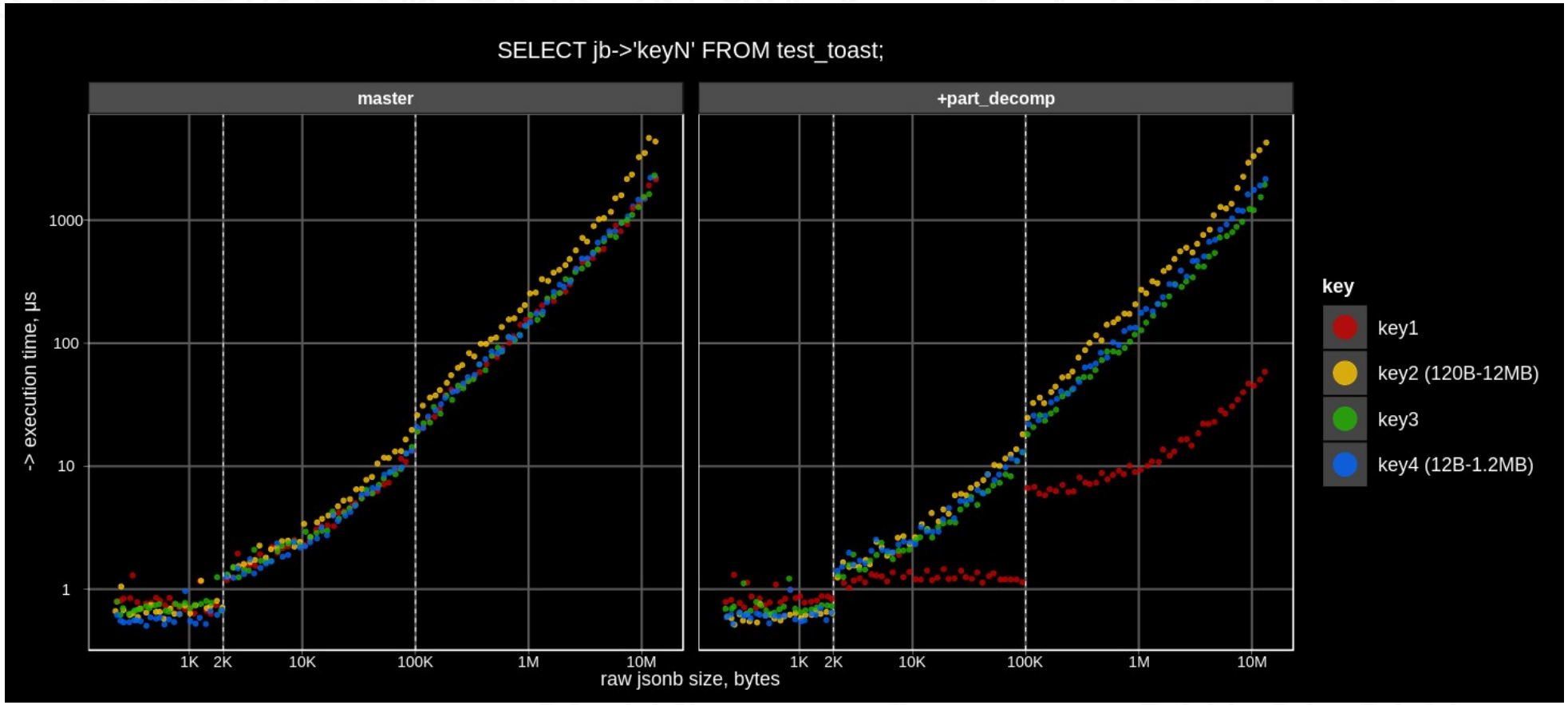

partial decompression

PostgresPro

# Jsonb partial decompression results (synthetic)

Access to key1 (red) in the prefix of jsonb was significantly improved:

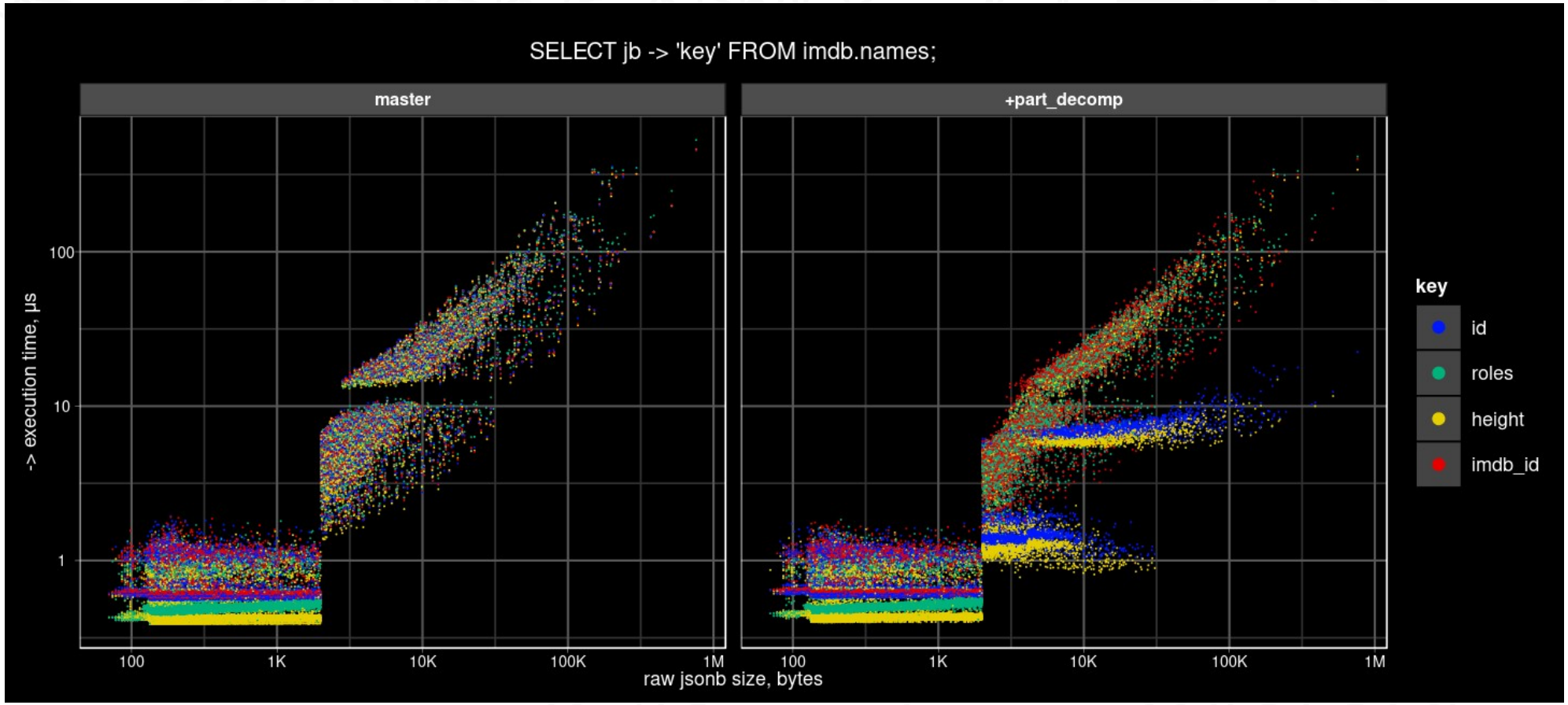
- For inline compressed jsonb access time becomes constant
- For jsonb > 1MB acceleration is of order(s) of magnitude.




PostgresPro

# Jsonb partial decompression results (IMDB)

- Access to the first key «id» and rare key «height» was significantly improved.
- Access time to big key «roles» and short «imdb_id» remains mostly unchanged




PostgresPro

# Sorting jsonb keys by length

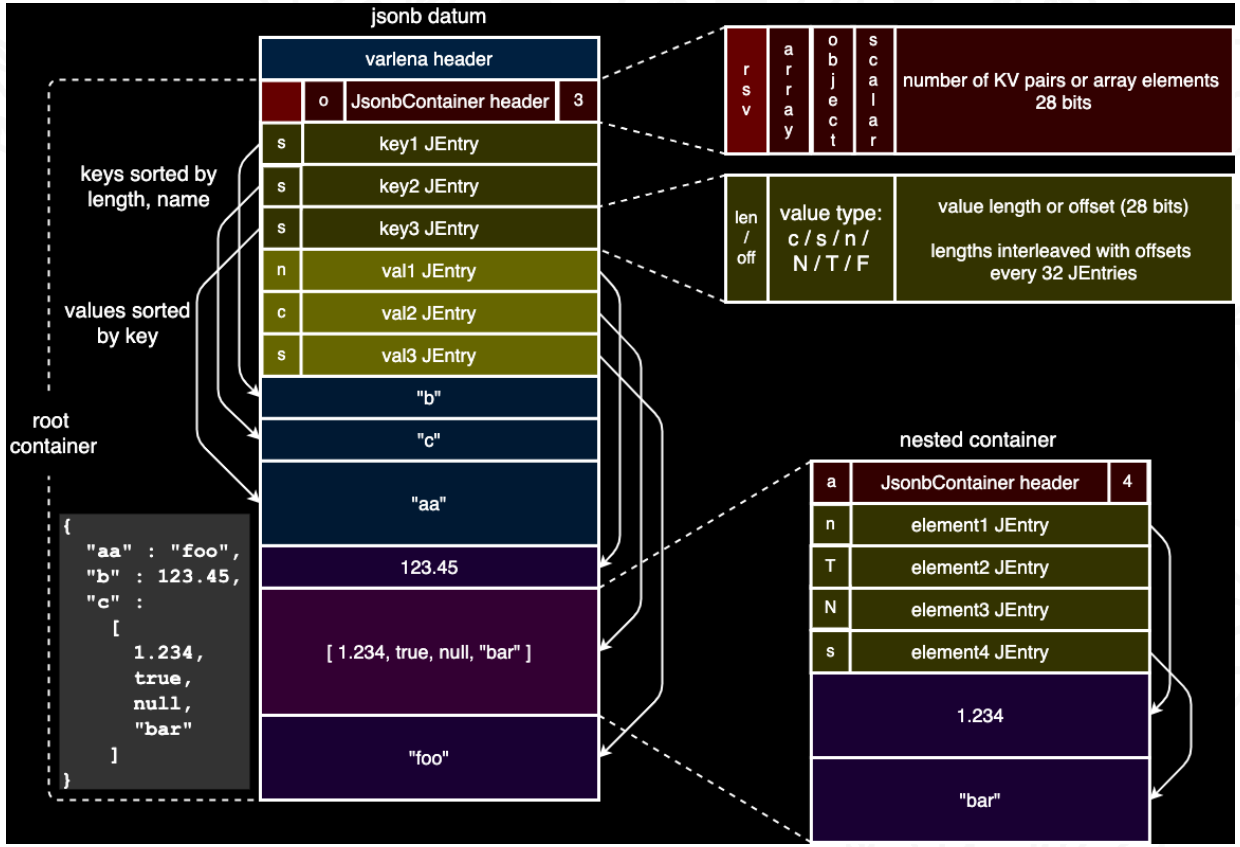
In the original jsonb format object keys are sorted by (length,name), so the short keys with longer or alphabetically greater names are placed at the end and cannot benefit from the partial decompression. Sorting by length allows fast decompressions of the shortest keys (metadata).

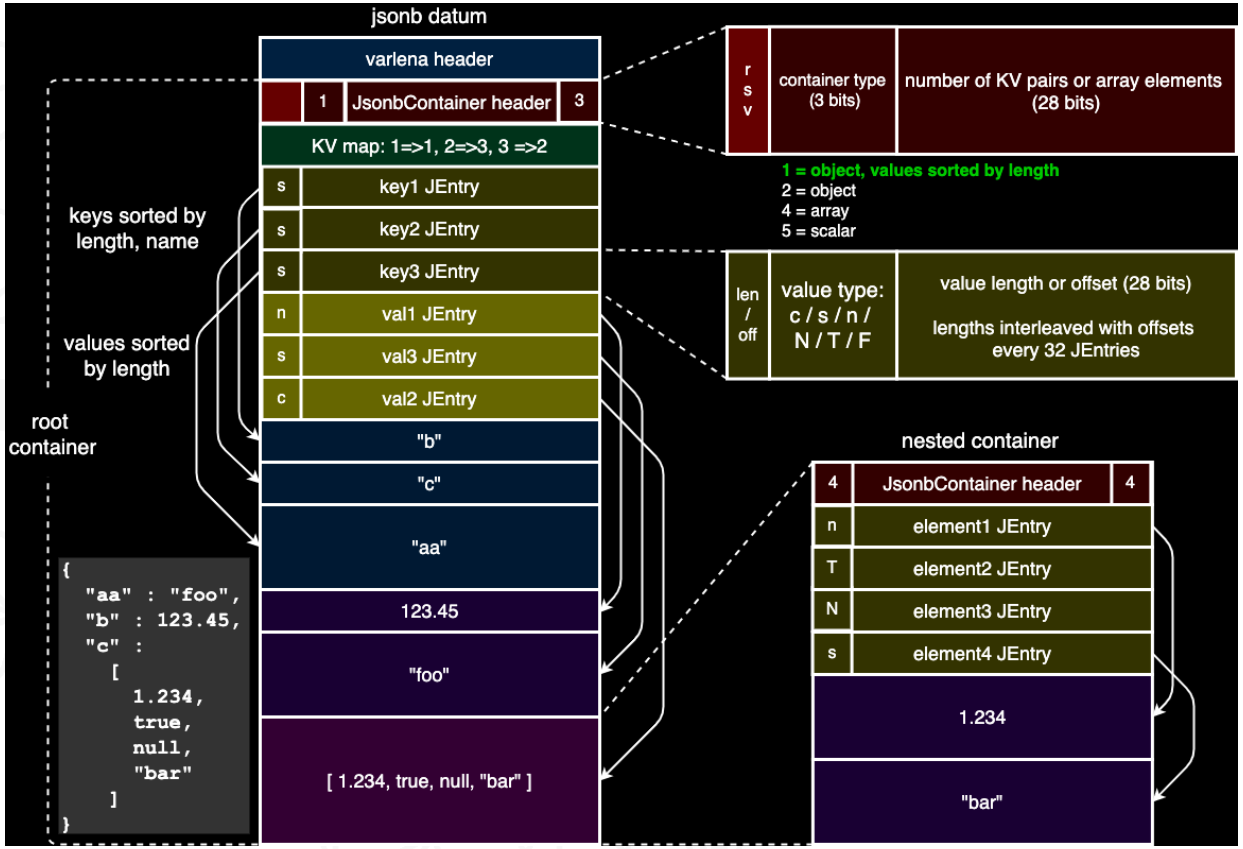

original: keys names and values sorted by key names


new: keys values sorted by their length

PostgresPro

# JSONB Binary Format (src/include/utils/jsonb.h)



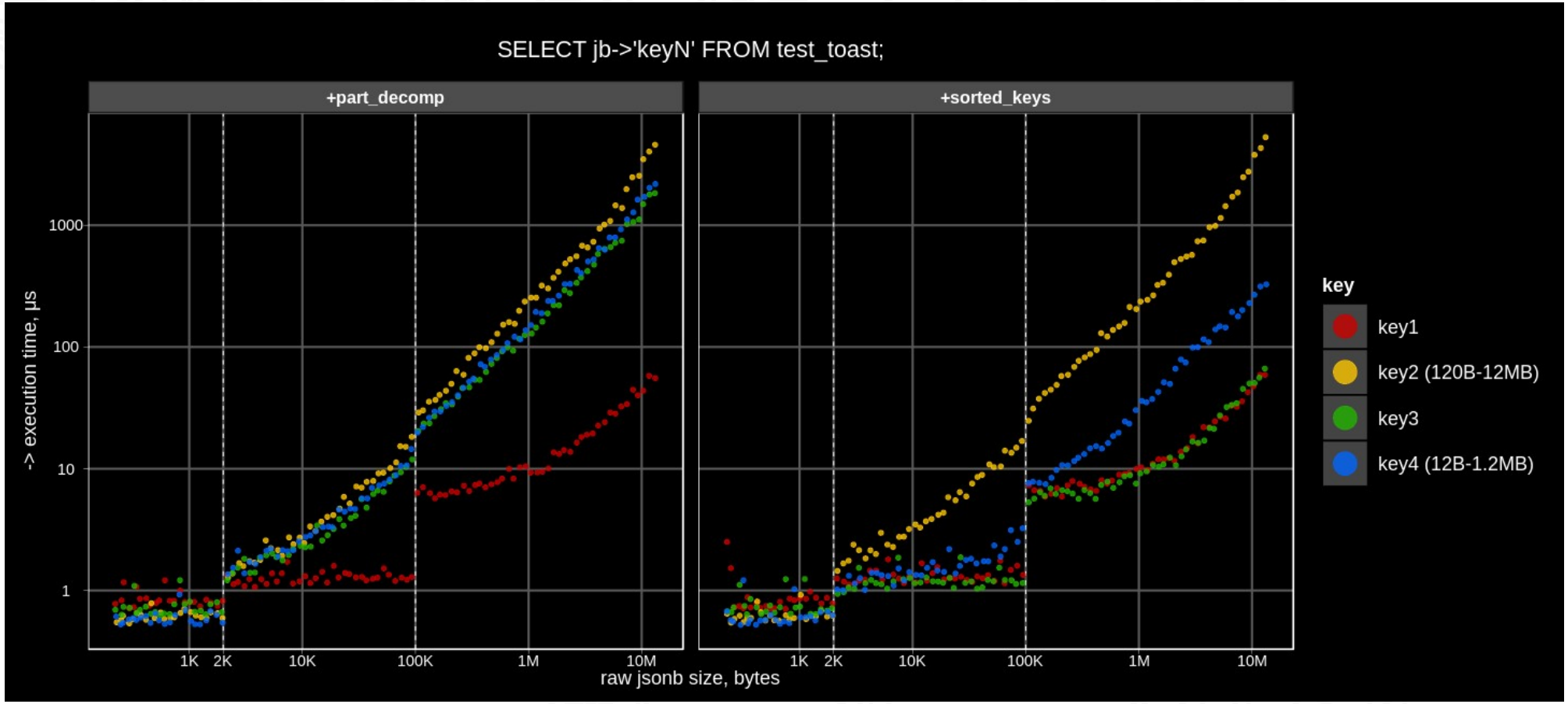

ORIGINAL: VALUES SORTED BY KEYS

VALUES SORTED BY THEIR LENGTH

PostgresPro

# Sorting jsonb keys by length results (synthetic)

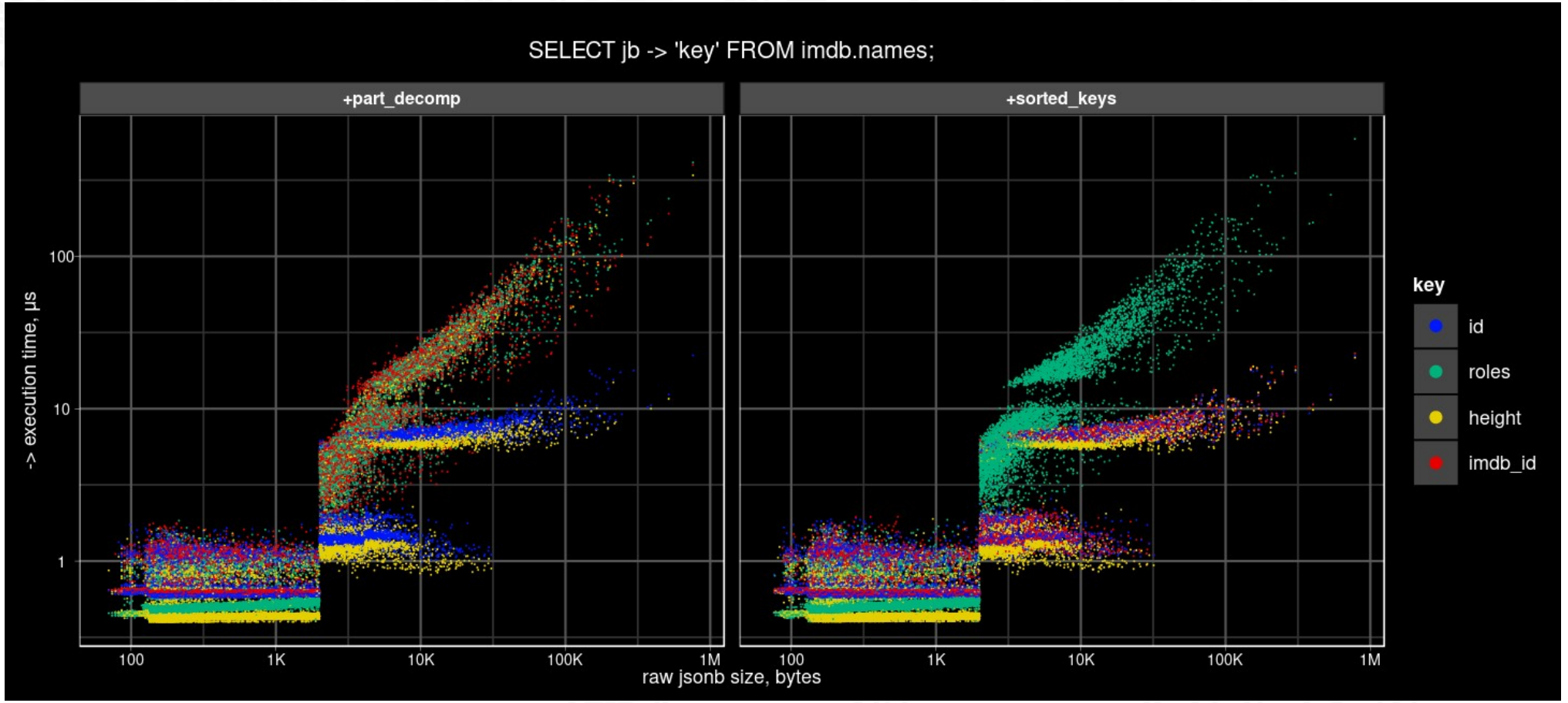
Access time to the all short keys and medium-length key4 (excluding long key2, placed now at the end of jsonb) was significantly speed up:




PostgresPro

# Sorting jsonb keys by length results (IMDB)

- Access to the last short key «imdb_id» now also was speed up.
- There is a big difference in access time (~5x) between inline and TOASTed values.




PostgresPro

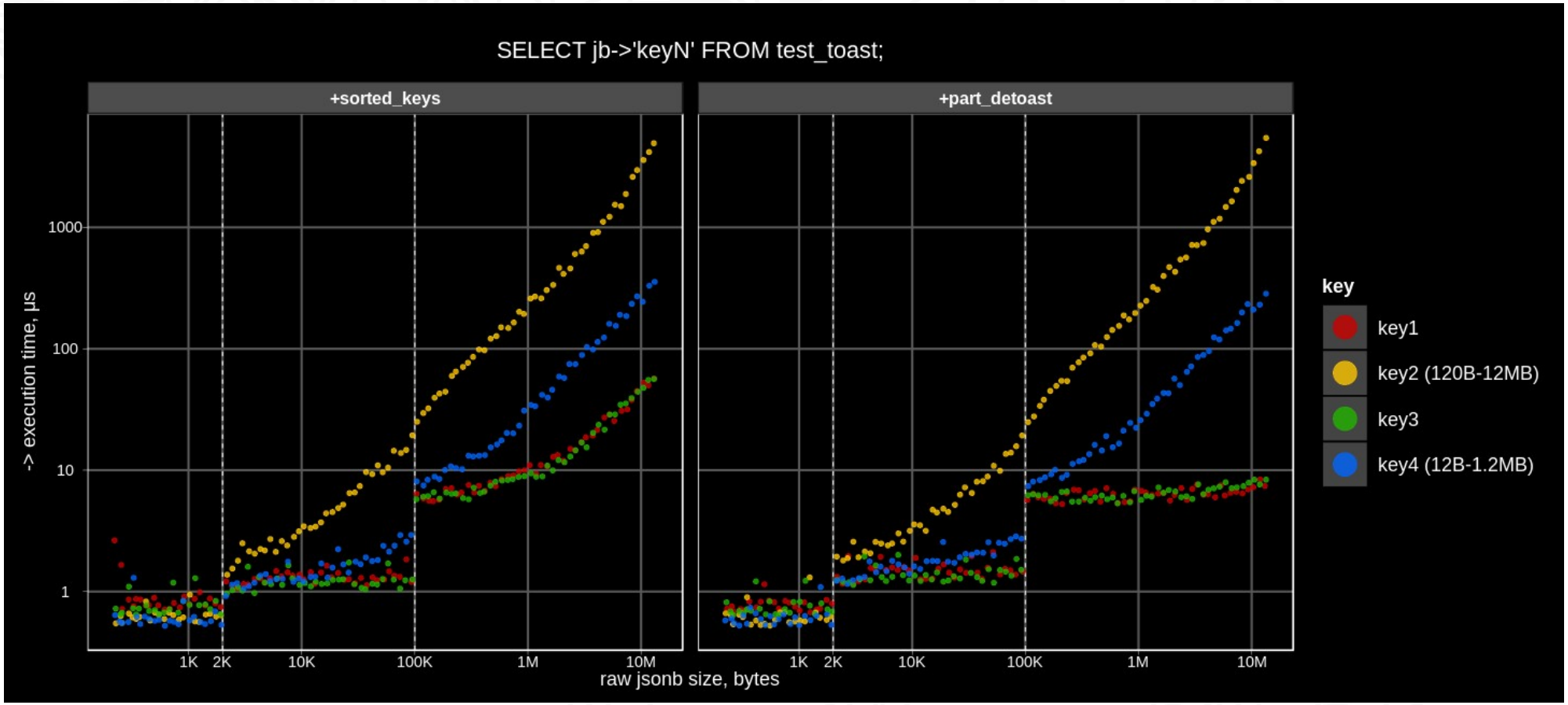
# Partial deTOASTing

- We used patch «de-TOAST'ing using a iterator» from the CommitFest. It was originally developed by Binguo Bao at GSOC 2019.
- This patch gives ability to deTOAST and decompress chunk by chunk. So if we need only the jsonb header and first keys from the first chunk, only that first chunk will be read (actually, some index blocks also will be read).
- We modified patch adding ability do decompress only the needed prefix of TOAST chunks.

PostgresPro

# Partial deTOASTing results (synthetic)

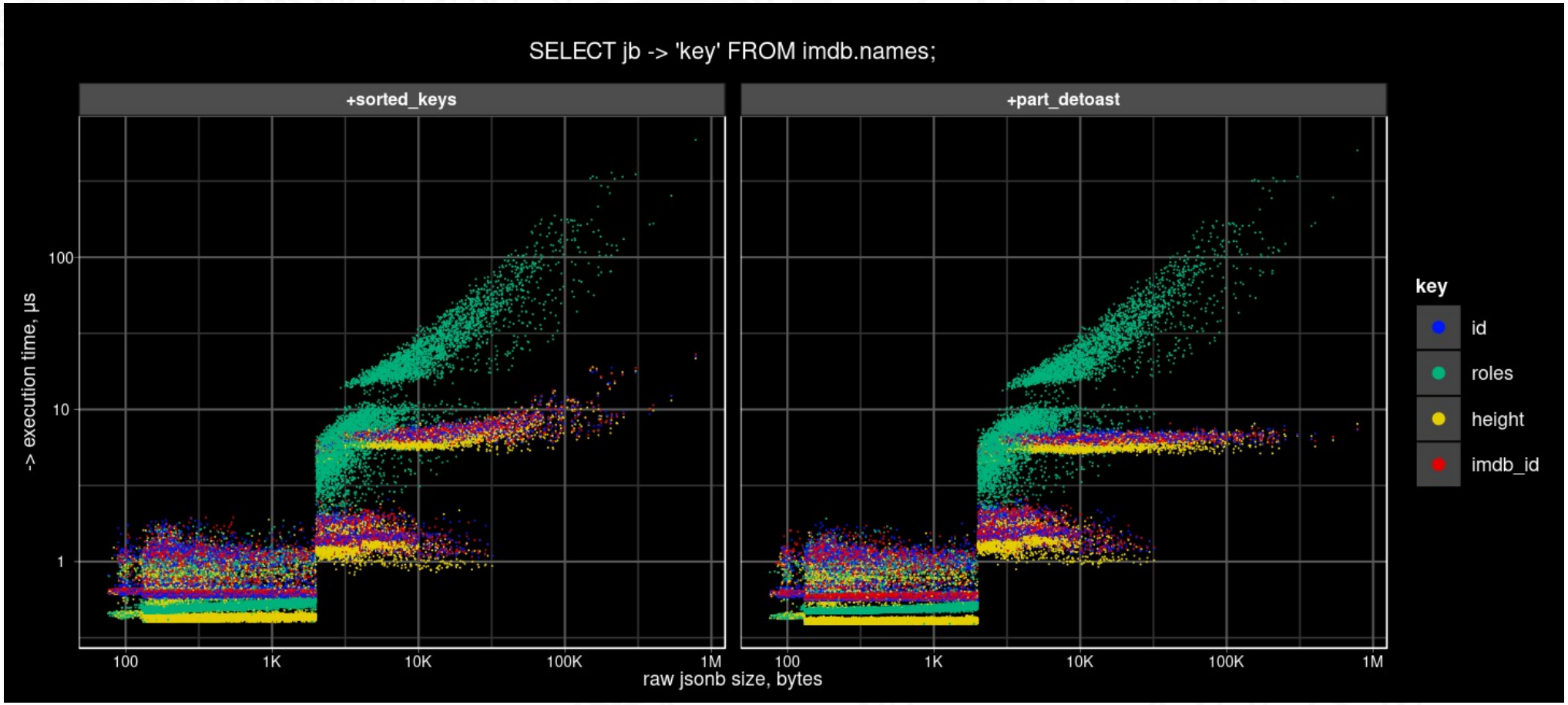
Partial deTOASTing speeds up only access to the short keys of long jsonbs, making access time almost independent of jsonb size.




PostgresPro

# Partial deTOASTing results (IMDB)

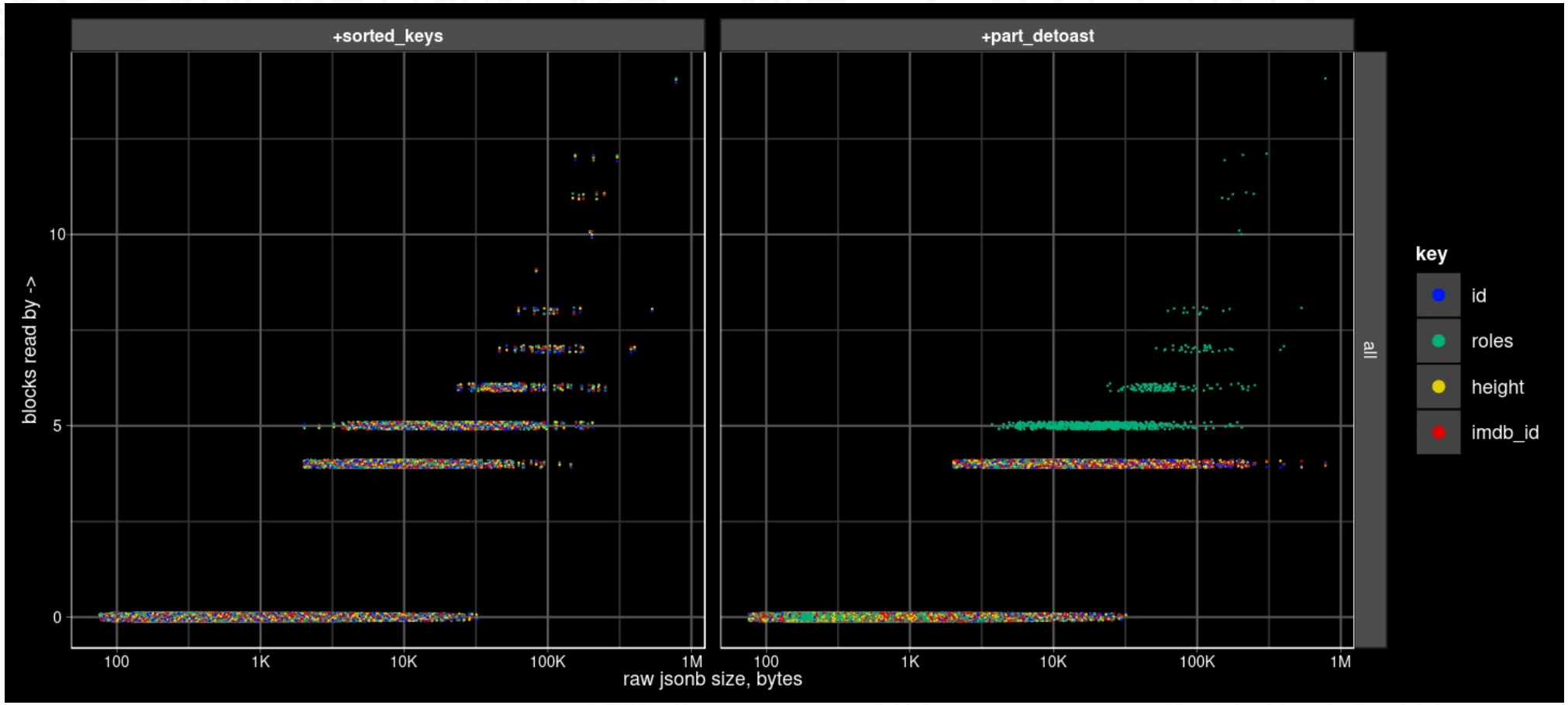
- Results are the same, but not so noticeable because the are not many big (> 100KB) jsonbs.
- A big gap in access time (~5x) between inline and TOASTed values is still there.




PostgresPro

# Partial deTOASTing results (IMDB)

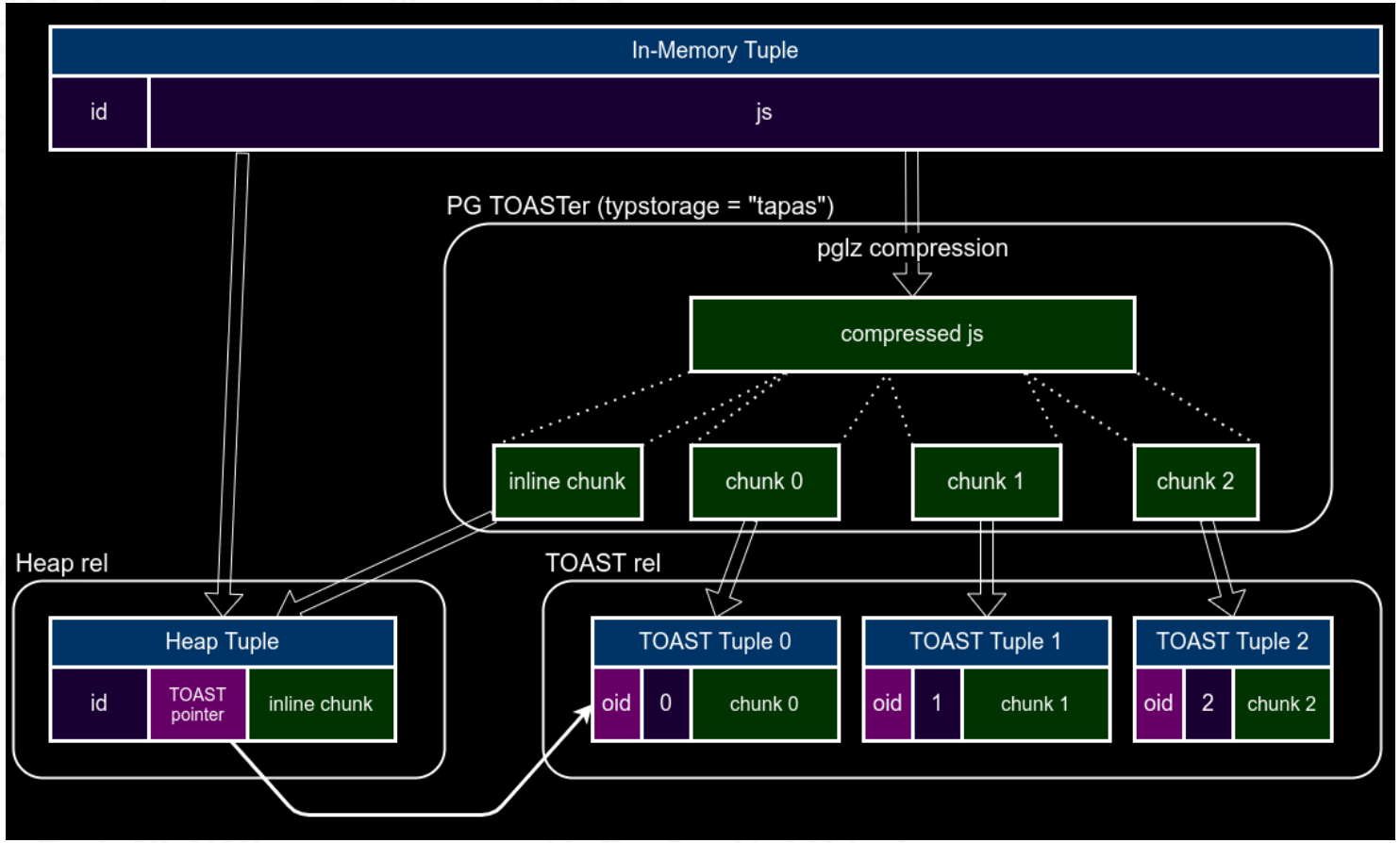
- Effect of partial deTOASTing : Arrow operator (→) for short keys always read only 4 blocks (3 index and 1 heap).




PostgresPro

# Iniline TOAST

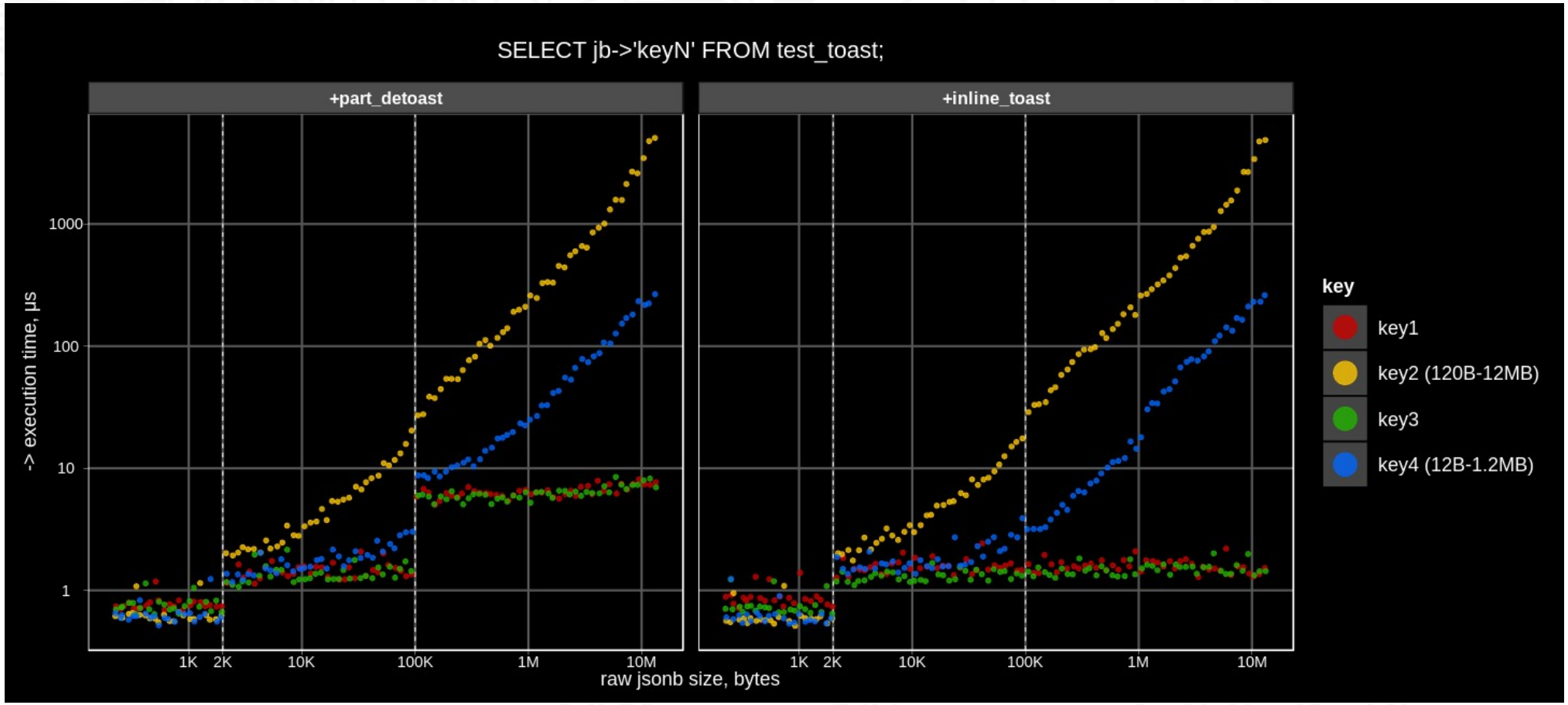
- Store first TOAST chunk containing jsonb header and possibly some short keys inline in the heap tuple.
- We added new typstorage «tapas», which is similar to «extended», except that it tries to fill the tuple to 2KB (if other attrubutes occupy less than 2KB) with the chunk cutted from the begining of the compressed data.




PostgresPro

# Inline TOAST results (synthetic)

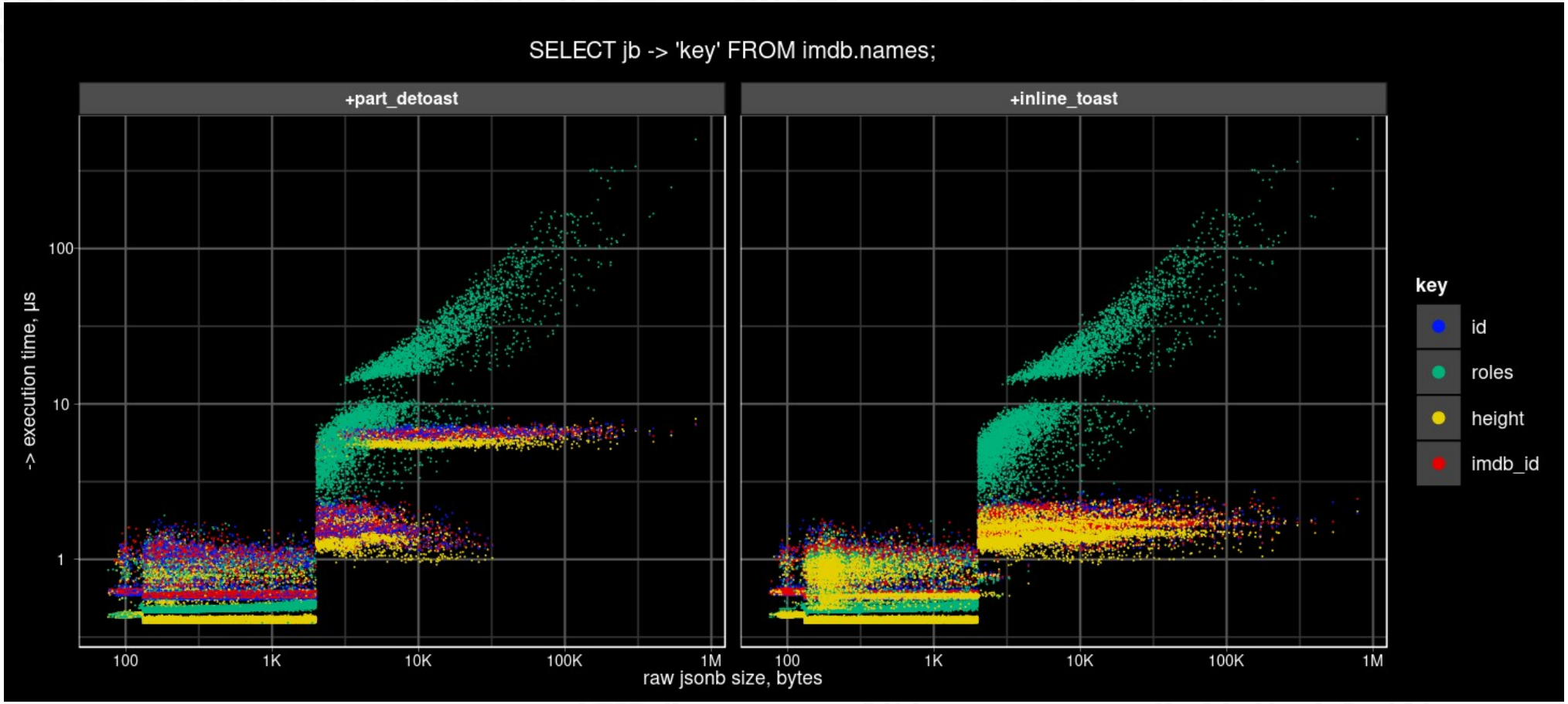
Partial inline TOAST completely removes gap in access time to short keys between long and mid-size jsonbs.




PostgresPro

# Inline TOAST results (IMDB)

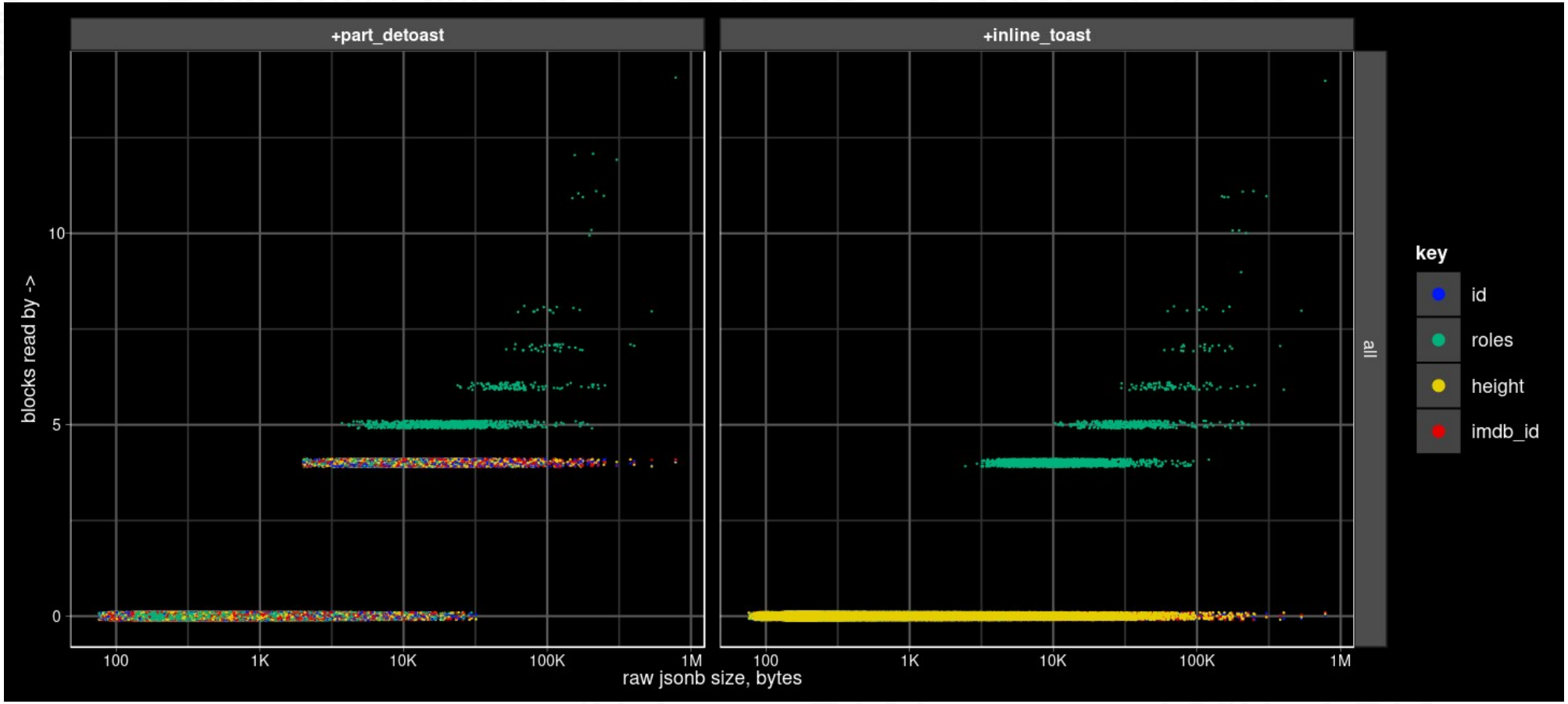
- Results are the same as in synthetic test.
- There is some access time gap between compressed and non-compressed jsonbs.




PostgresPro

# Inline TOAST results (IMDB)

- Effect of inline TOAST :  Arrow operator (→) for short keys  read no additional blocks.




PostgresPro

# JSONB partial update

TOAST was originally designed for atomic data types, it knows nothing about internal structure of composite data types like jsonb, hstore, and even ordinary arrays.

TOAST works only with binary BLOBs, it does not try to find differencies between old and new values of updated attributes. So, when the TOASTed attribute is being updated (does not matter at the beginning or at the end and how much data is changed), its chunks are simply fully copied. The consequences are:

- TOAST storage is duplicated
- WAL traffic is increased in comparison with updates of non-TOASTED attributes, because the whole TOASTed values is logged
- Performance is too low

PostgresPro

# JSONB partial update: The problem

Example: table with 10K jsonb objects with 1000 keys `{ "1": 1, "2": 2, ... }`.

```
CREATE TABLE t AS
SELECT i AS id, (SELECT jsonb_object_agg(j, j) FROM generate_series(1, 1000) j) js
FROM generate_series(1, 10000) i;

SELECT oid::regclass AS heap_rel,
       pg_size_pretty(pg_relation_size(oid)) AS heap_rel_size,
       reltoastrelid::regclass AS toast_rel,
       pg_size_pretty(pg_relation_size(reltoastrelid)) AS toast_rel_size
FROM pg_class WHERE relname = 't';

 heap_rel | heap_rel_size |       toast_rel        | toast_rel_size
----------+---------------+------------------------+----------------
 t        | 512 kB        | pg_toast.pg_toast_27227 | 78 MB
```

Each 19 KB jsonb is compressed into 6 KB and stored in 4 TOAST chunks.

```
SELECT pg_column_size(js) compressed_size, pg_column_size(js::text::jsonb) orig_size from t limit 1;
 compressed_size | original_size
-----------------+---------------
            6043 |         18904

SELECT chunk_id, count(chunk_seq) FROM pg_toast.pg_toast_47235 GROUP BY chunk_id LIMIT 1;
 chunk_id | count
----------+-------
    57241 |     4
```

PostgresPro

# JSONB partial update: The problem

First, let's try to update of non-TOASTED int column `id`:

```
SELECT pg_current_wal_lsn(); --> 0/157717F0

UPDATE t SET id = id + 1; -- 42 ms

SELECT pg_current_wal_lsn(); --> 0/158E5B48

SELECT pg_size_pretty(pg_wal_lsn_diff('0/158E5B48','0/157717F0')) AS wal_size;
 wal_size
----------
 1489 kB   (150 bytes per row)

SELECT oid::regclass AS heap_rel,
       pg_size_pretty(pg_relation_size(oid)) AS heap_rel_size,
       reltoastrelid::regclass AS toast_rel,
       pg_size_pretty(pg_relation_size(reltoastrelid)) AS toast_rel_size
FROM pg_class
WHERE relname = 't';
 heap_rel | heap_rel_size |        toast_rel         | toast_rel_size
----------+---------------+--------------------------+---------------
 t        | 1024 kB       | pg_toast.pg_toast_47235 | 78 MB
          (was 512 kB)                                (not changed)
```

PostgresPro

# JSONB partial update: The problem

Next, let's try to update of TOASTED jsonb column `js`:

```
SELECT pg_current_wal_lsn(); --> 0/158E5B48

UPDATE t SET js = js - '1'; -- 12316 ms (was 42 ms, ~300x slower)

SELECT pg_current_wal_lsn(); --> 0/1DB10000

SELECT pg_size_pretty(pg_wal_lsn_diff('0/1DB10000','0/158E5B48')) AS wal_size;
 wal_size
----------
 130 MB    (13 KB per row; was 1.5 MB, ~87x more)

SELECT oid::regclass AS heap_rel,
       pg_size_pretty(pg_relation_size(oid)) AS heap_rel_size,
       reltoastrelid::regclass AS toast_rel,
       pg_size_pretty(pg_relation_size(reltoastrelid)) AS toast_rel_size
FROM pg_class
WHERE relname = 't';
 heap_rel | heap_rel_size |        toast_rel         | toast_rel_size
----------+---------------+--------------------------+---------------
 t        | 1528 kB       | pg_toast.pg_toast_47235 | 156 MB
           (was 1024 kB)                             (was 78 MB, 2x more)
```

PostgresPro

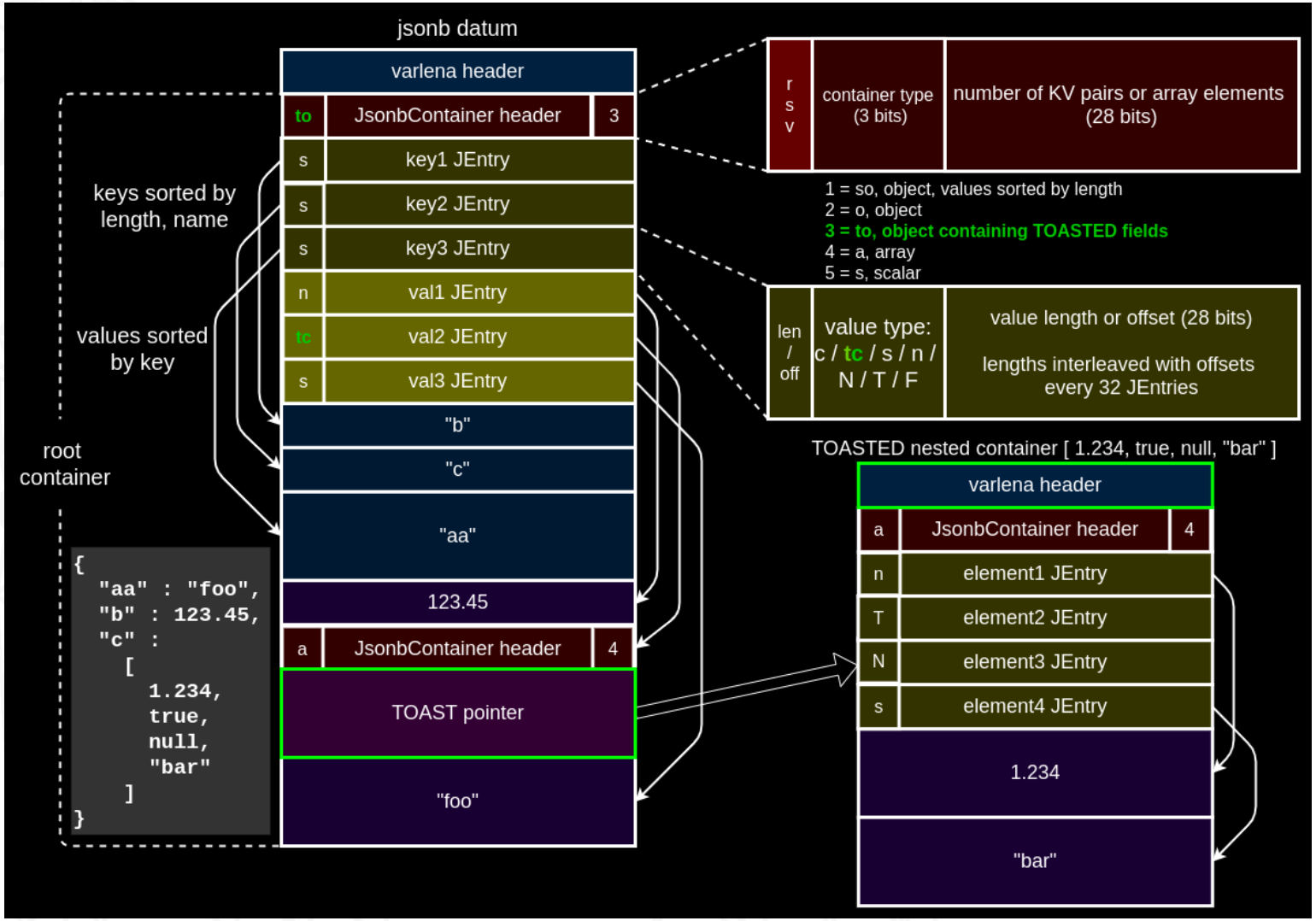
# Partial update using Shared TOAST

- The previsous optimizations are great for SELECT, but don't help with UPDATE, since TOAST consider jsonb as an atomic binary blob – change part, copy the whole.
- Idea: keep container header together with short fields and long fields replaced by TOAST pointers INLINE, so the container headers and short fields can be updated without copying of TOASTed fields, which will be shared.
- Currently, this works only for root objects fields, so the longest fields of jsonb object are TOASTed until the whole tuple fits into the page (typically, remaining size of jsonb becomes < ~2000 bytes).
- But this technique can also be applied to array elements or element ranges. We plan to try to implement it later, it needs more invasive jsonb API changes.
- Now jsonb hook is hardcoded into TOAST pass #1, but in the future it will become custom datatype TOASTer using `pg_type.typtoast`.

PostgresPro

# Shared TOAST – jsonb format extensions

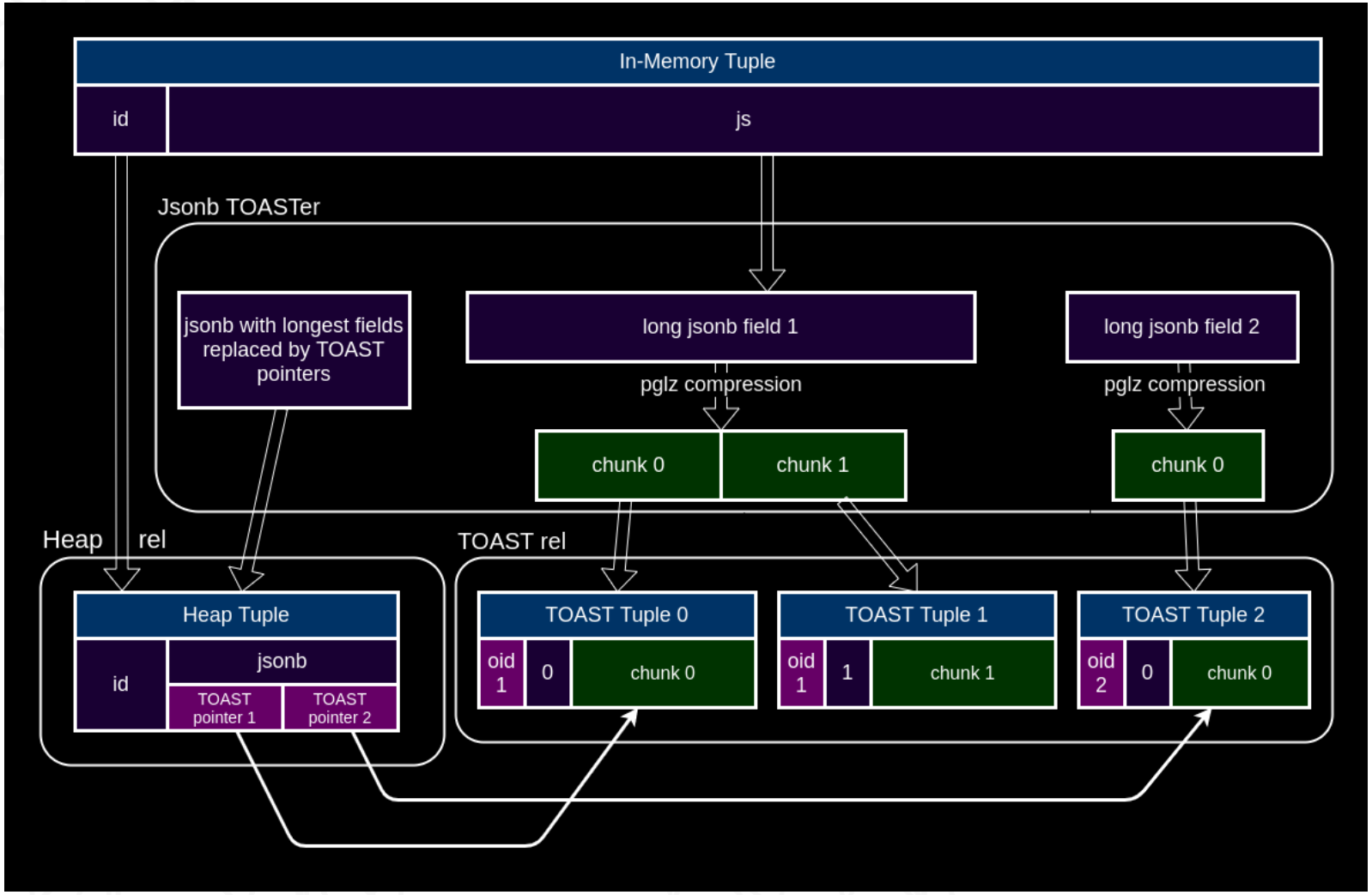
- Added special “TOASTed container” `JEntry` type. `JsonbContainer` header is left inline, but the body is replaced with a pointer.
- Added “TOASTed object” `JsonbContainer` type to mark object with TOAST pointers.
- TOASTed subcontainers are stored as plain jsonb datums (`varlena` header added).




PostgresPro

# Shared TOAST – tuple structure

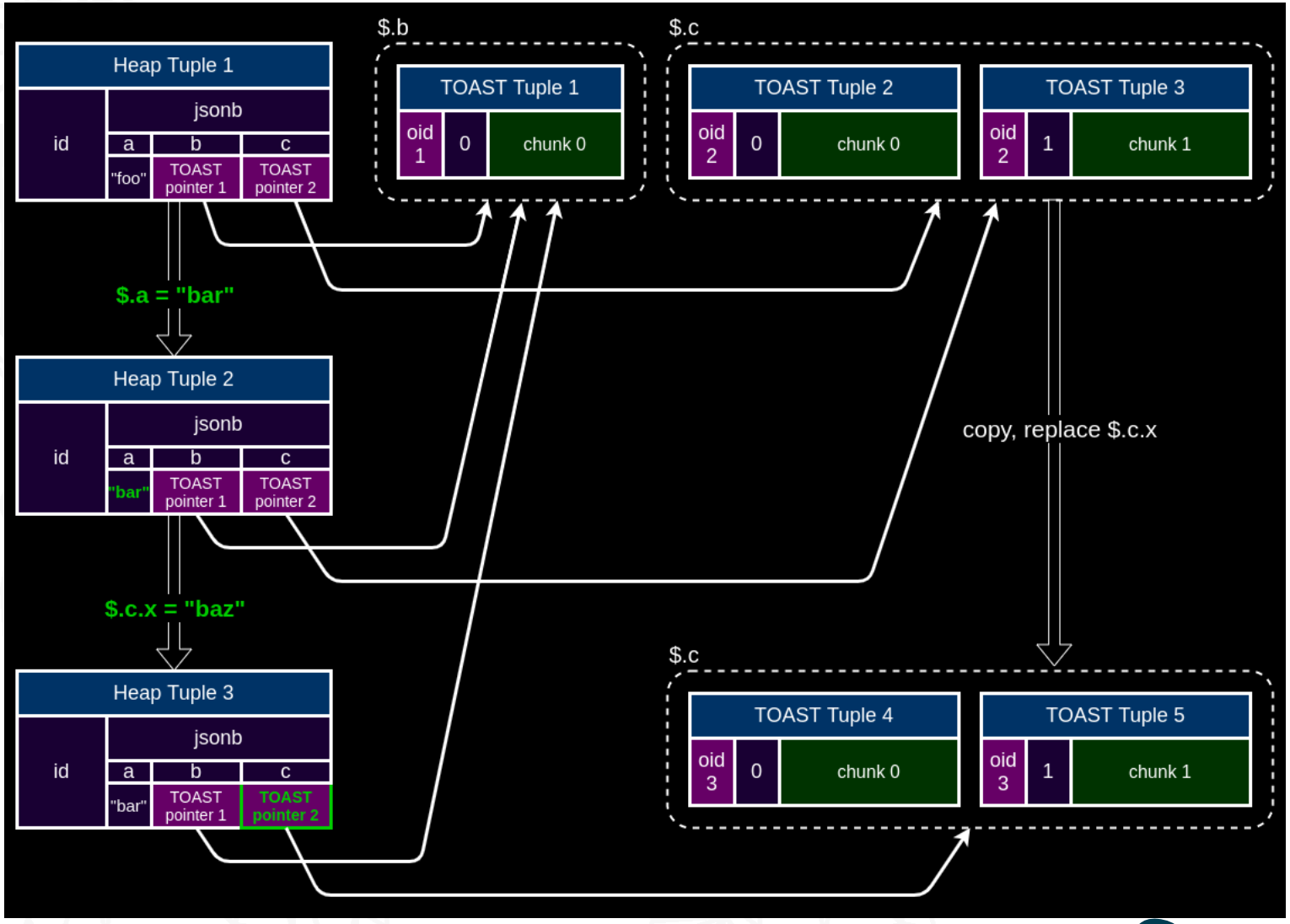
- In this example two longest fields of jsonb are TOASTed separately
- TOASTed jsonb contains two TOAST pointers
- Operators like -> can simply return TOAST pointer as external datum, accessing only the inline part of jsonb




PostgresPro

# Shared TOAST – update

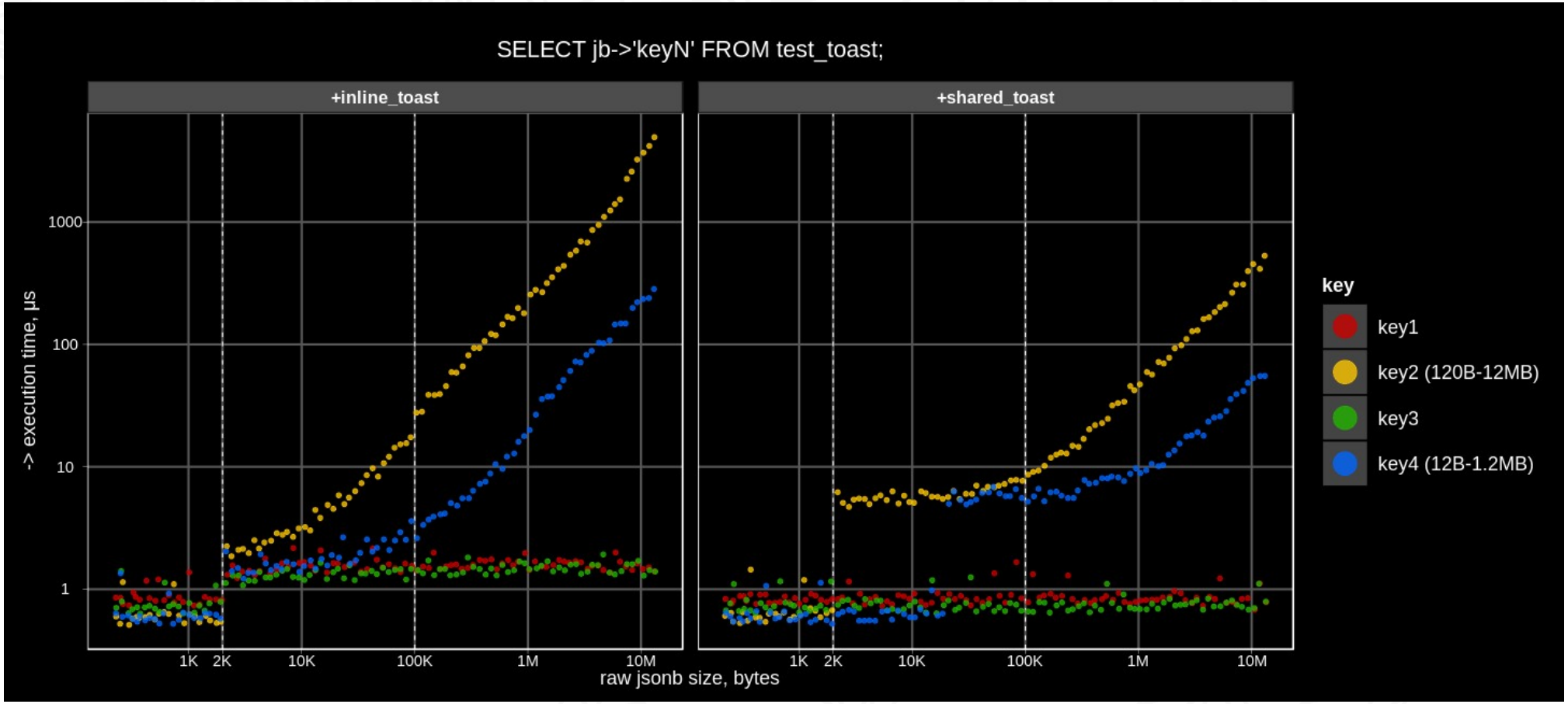
- When the short inline field is updated, only the new version of inline data is created.
- When some part of long the long field is updated, the whole container is copied, updated and then TOASTed back with new oid (in the future oids can be shared).
- Unchanged TOASTed fields are always shared.




PostgresPro

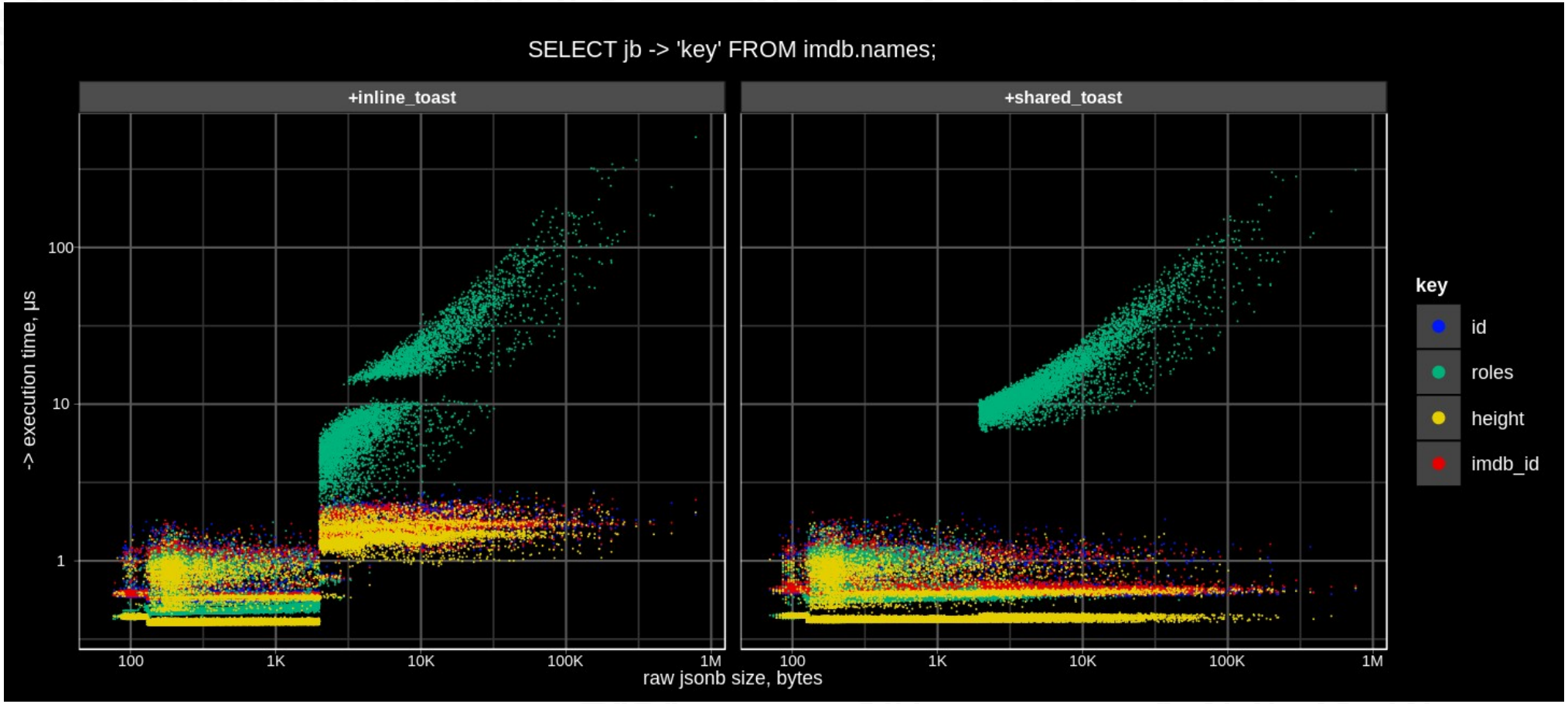
# Shared TOAST – access results (synthetic)

Gap in access time to short keys is completely removed. Access to mid-size fields is slow down, because they are TOASTed instead of stored inline (we need to fix this).



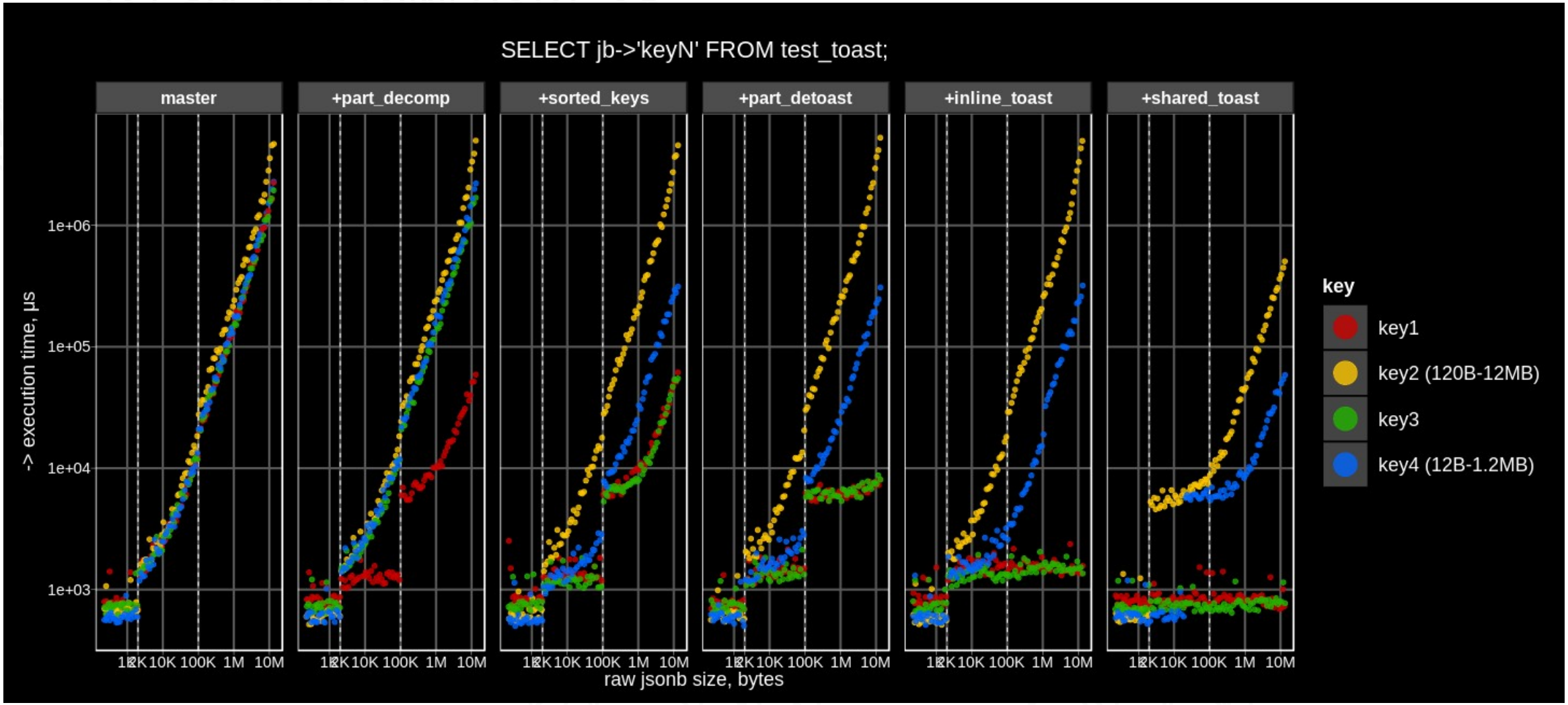
# Shared TOAST – access results (IMDB)

- Results are the same as in synthetic test.
- All short keys is speed up as much as possible.



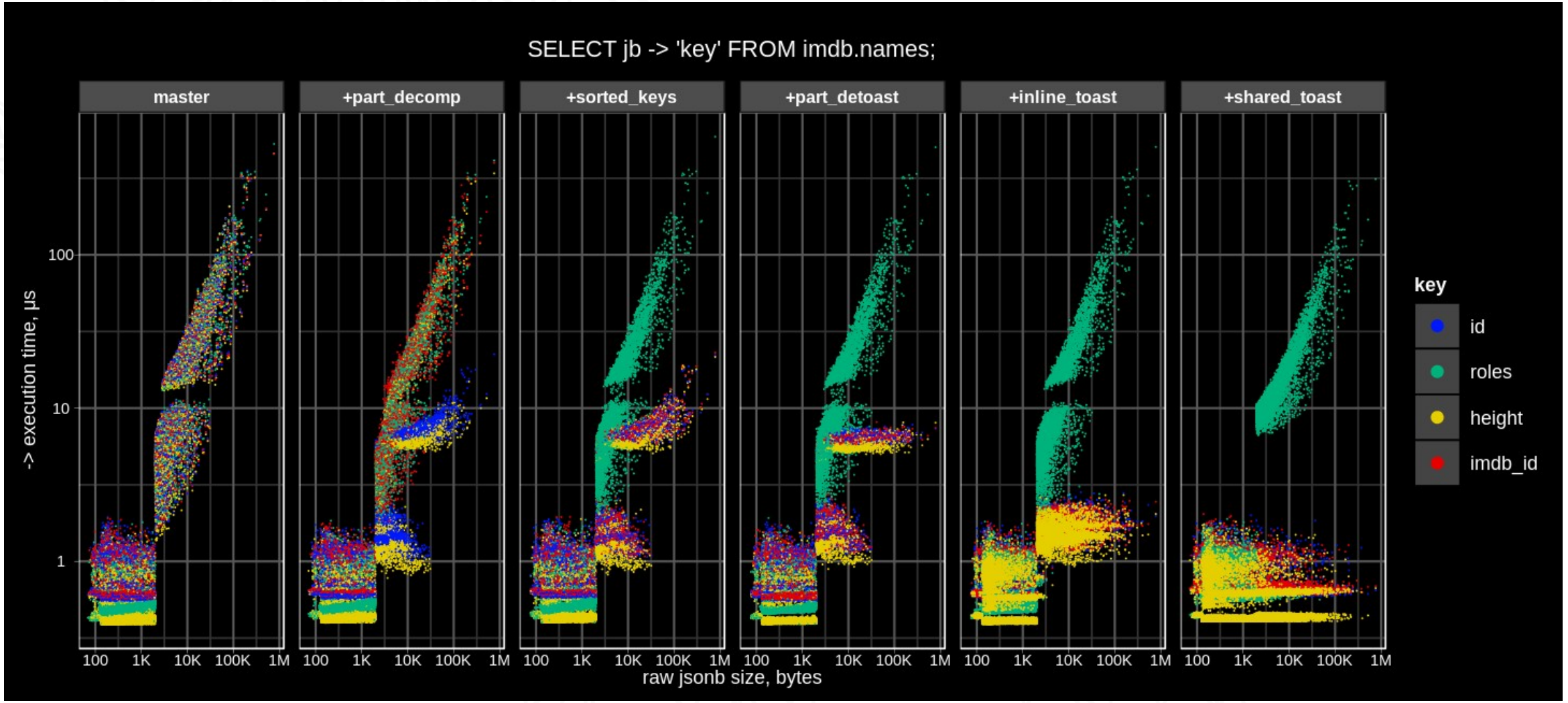

PostgresPro

# Step-by-step results (synthetic)




PostgresPro

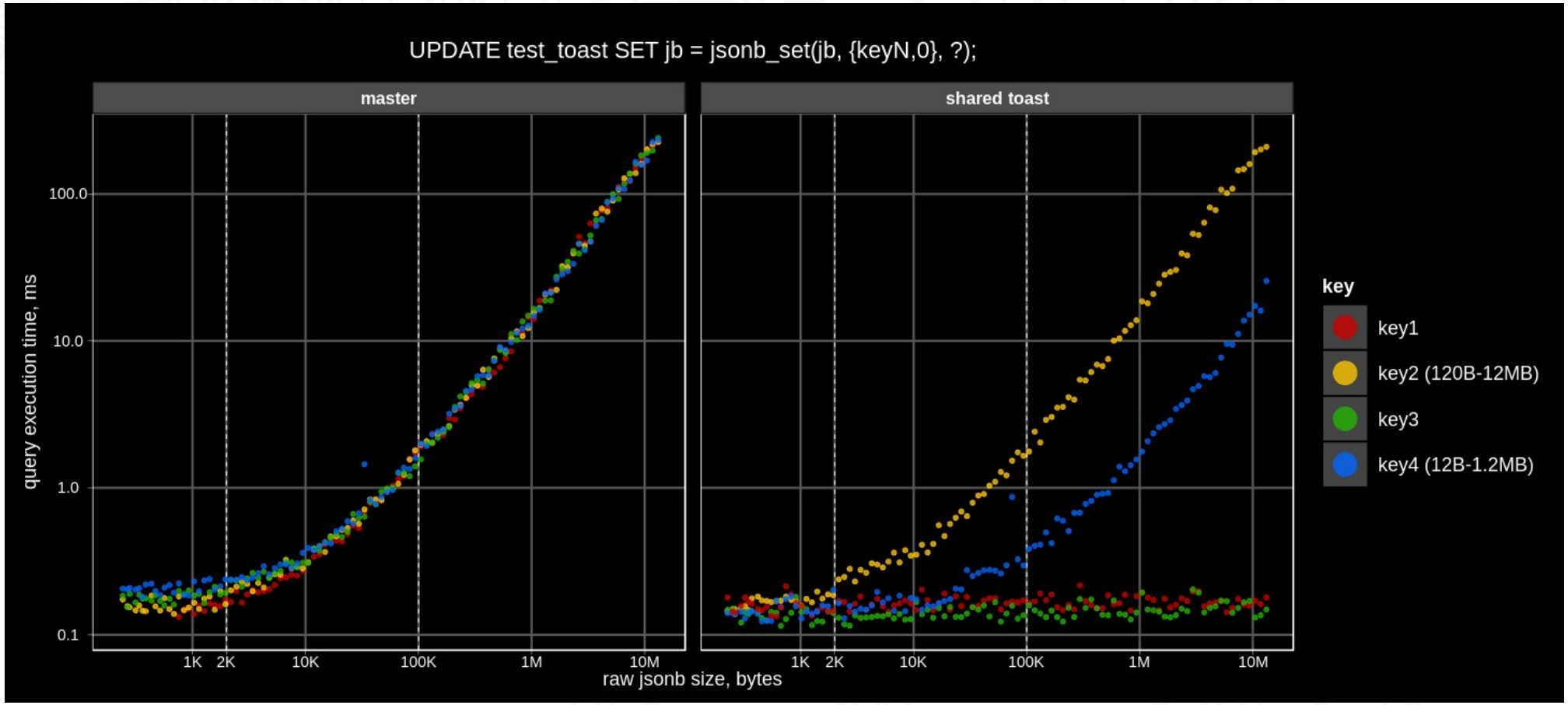
# Step-by-step results (IMDB)




PostgresPro

# Shared TOAST – update results (synthetic)

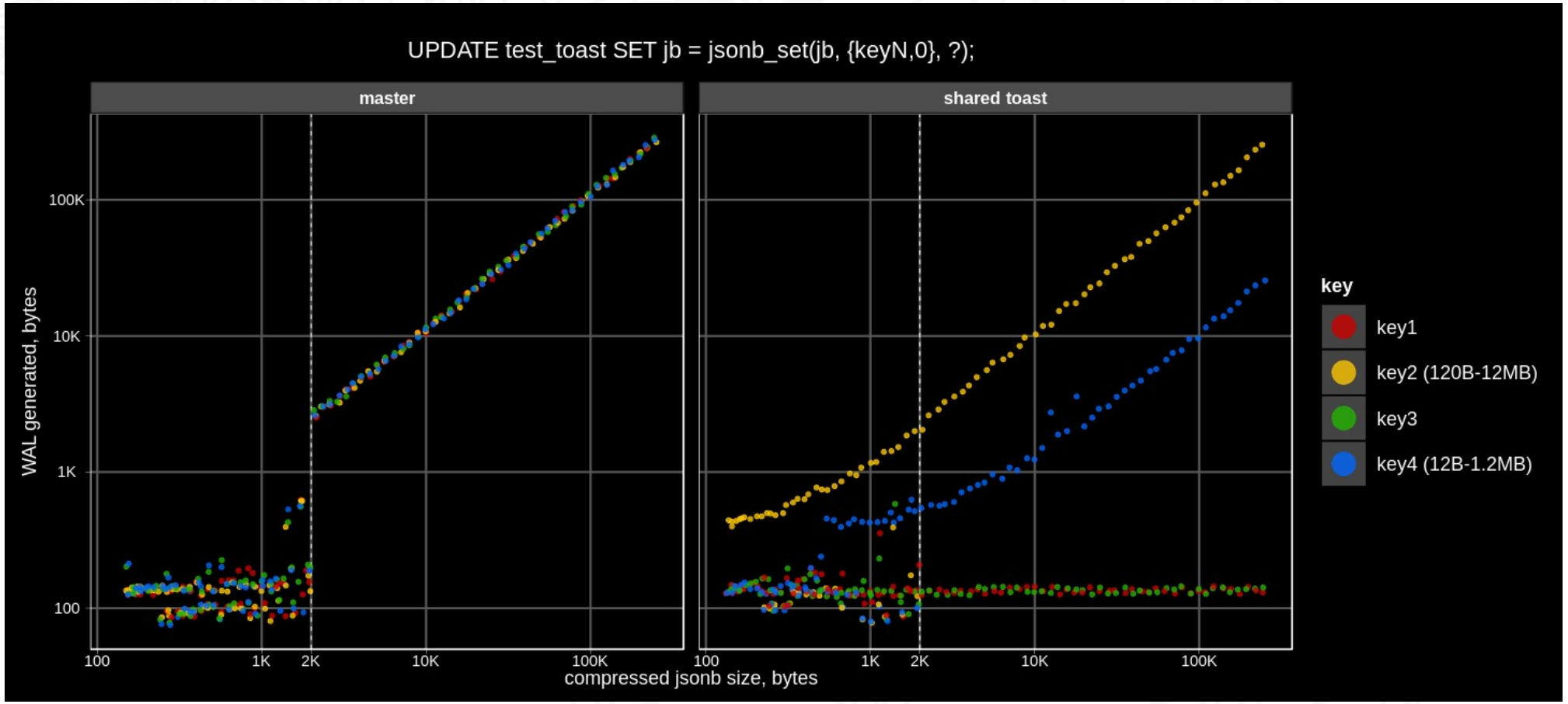
- Update time of short keys does not depend on total jsonb size
- Update time of TOASTed fields depends only on their own size




PostgresPro

# Shared TOAST – update results (synthetic)

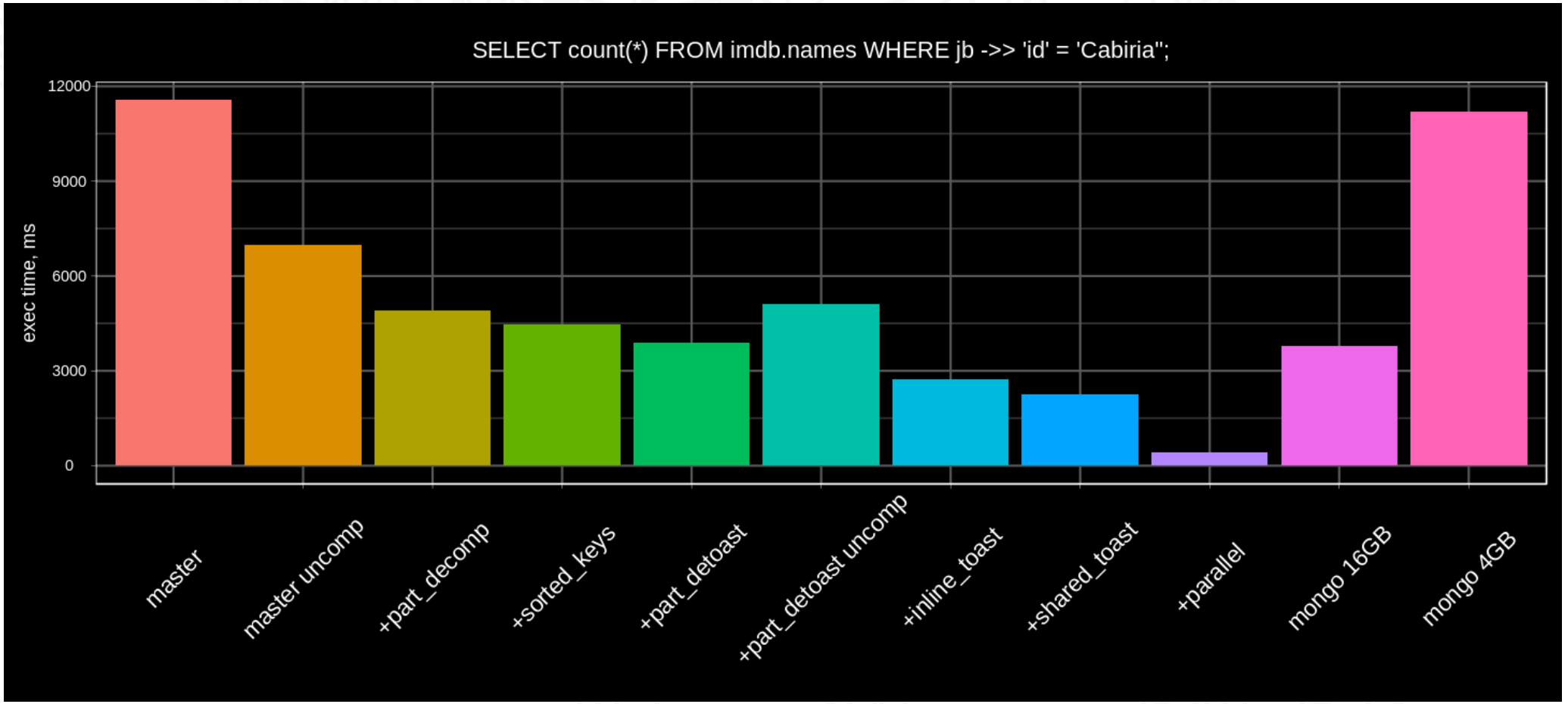
- WAL traffic due to update of short and mid-size keys is greatly decreased




PostgresPro

# Non-scientific comparison PG vs Mongo

- Seqscan, PG - in-memory, Mongo (4.4.4): 16Gb (in-memory), 4GB (1/2)




PostgresPro

# Quick Summary and references

- We demonstrated step-by-step performance improvements (with backward compatibility), which lead to significant (10X) speedup for SELECTs and much cheaper UPDATEs (OLTP Jsonb?)
  - Github: https://github.com/postgrespro/postgres/tree/jsonb_shared_toast
  - Slides of this talk (PDF, Video)
  - It's not PG14 ready
- The same technique can be applied to any data types with random access to parts of data (arrays, hstore, movie, pdf …)
- Jsonb is ubiquitous and is constantly developing
  - JSON[B] Roadmap V2, Postgres Professional Webinar, Sep 17, 2020
  - JSON[B] Roadmap V3, Postgres Build 2020, Dec 8, 2020
- Contact obartunov@postgrespro.ru, n.gluhov@postgrespro.ru for collaboration.

PostgresPro

# TODO (OLTP JSONB, OLAP JSONB)

- Optimization of GSON code to eliminate overhead
- More benchmarks (YCSB, use cases),  PG vs Mongo
- Extend shared TOAST to support arrays, strings
- How to integrate this new stuff into the CORE ?
- WiredElephant – storage (non-TOASTED)  for tree-like structures with big attributes ?
- TOAST cache - to avoid duplication of deTOASTing, if the query contains two or more jsonb operators and function on the the same jsonb attribute.
- DeTOAST deferring in the chain of accessors (`js→'roles'→5`), not needed for jsonpath.

PostgresPro

Нам нужны Ваши кейсы (тестовые данные и запросы) !

PostgresPro


ALL
YOU
NEED
POSTGRES
IS